TAMPEREEN TEKNILLINEN YLIOPISTO

TERO LAHTINEN

FUNKTIONAALISTEN JA OLIO-IMPERATIIVISTEN OHJELMISTOKOMPONENTTIEN YHDISTÄMINEN

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston
kokouksessa 14.8.2013

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TERO LAHTINEN: Funktionaalisten ja olio-imperatiivisten ohjelmistokomponenttien yhdistäminen
Diplomityö, 66 sivua, 15 liitesivua
Toukokuu 2014
Pääaine: Ohjelmlistotuotanto
Tarkastajat: professori Tommi Mikkonen
Avainsanat: paradigma, olio-ohjelmointi, funktionaalisuus, C#, F#

Yksi ensimmäisistä tehtävistä ohjelman toteutuksen alkuvaiheessa on ohjelmointikielen valinta. Tutkimukset ovat osoittaneet, että sopivan ohjelmointikielen valinnalla on suuri merkitys ohjelman elinkaarikustannuksiin. Siksi valinta tulisikin tehdä hyvin perusteltujen syiden pohjalta.

Koska yleisimmin käytetyt ohjelmointikielet ovat olio-imperatiivisia, on siksi olemassa erittäin paljon valmiita ohjelmistokomponentteja, jotka on toteutettu näillä kielillä. Täysin uuttakaan ohjelmaa toteutettaessa ei yleensä haluta toteuttaa kaikkia ominaisuuksia aivan alusta asti, vaan ohjelmiston suunnittelussa pyritään hyödyntämään olemassa olevia komponentteja. Kun ohjelmointikieleksi valitaan esimerkiksi funktionaalinen kieli, on usein varauduttava liittymään jollain muulla ohjelmointikielellä toteutettuun komponenttiin. Tällaisissa tapauksissa kahden erilaisen ohjelmointikielen ja niiden paradigmojen yhdistäminen saattaa olla haasteellista.

Tässä diplomityössä tutkitaan, miten C#:lla ja F#:lla toteutettujen komponenttien yhdistäminen voidaan tehdä ja miten eri paradigmojen yhdistämisestä seuraavat ongelmat voidaan ratkaista. Tämän lisäksi työssä tutkittiin C#:n ja F#:n käytettävyyttä haastattelututkimuksen avulla. Tutkimuksella selvitettiin, millaisia etuja funktionaalisen F#:n käytöllä on verrattuna olio-imperatiiviseen C#:iin ja saadaanko sen käytöstä niin paljon hyötyä, että sitä kannattaa käyttää vaikka joutuisi tekemään lisätyötä C#:lla toteutettuun ohjelmaan liittyäkseen.

Tämä diplomityö on tehty Atostek Oy:lle, jossa C#- ja F#-kieliä on hyödynnetty useissa käytännön ohjelmistoprojekteissa. Siinä missä C# on yleisesti käytetty olio-imperatiivinen kieli, on F# funktionaalinen. C#:n ja F#:n tapauksessa yhteinen ajoympäristö, Microsoftin .NET, ratkaisee ison osan eri kielillä kirjoitettujen komponenttien yhdistämisen ongelmista. Ratkaisematta kuitenkin jää edelleen paradigmojen yhdistämisestä aiheutuvat haasteet.

Haastattelututkimuksella onnistuttiin saamaan selville ohjelmistosuunnittelijoiden kummankin kielen käytännön hyödyntämiseen perustuvat mielipiteet. Haastattelujen tulokset ovat keskenään hyvin yhdenmukaisia ja antavat hyvän kuvan funktionaalisen F#:n hyödyntämisestä käytännössä.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TERO LAHTINEN : Combining functional and object-imperative software components
Master of Science Thesis, 66 pages, 15 Appendix pages
May 2014
Major: Software Engineering
Examiner: Professor Tommi Mikkonen
Keywords: paradigm, object oriented programming, functional programming, C#, F#

One of the first tasks when starting the implementation of a new program is to choose a programming language. Studies show that software life cycle cost can be influenced by choosing an appropriate programming language. Therefore the selection of the language should be based on rational reasoning.

The most popular programming languages today are object-imperative. Therefore there exist plenty of ready-made software components implemented in those languages. Even when implementing a completely new program, it is not usually appropriate to implement all its features from scratch. Therefore, ready-made components are used. When, for example, a functional programming language is chosen as the implementation language, the program must be prepared to be integrated to components implemented in some other language. In these cases integrating different languages and different paradigms may raise problems.

This master's thesis describes how software components implemented with C# and F# can be integrated, and how the problems caused by the different programming paradigms can be solved. Also the usability of C# and F# is studied by interviewing software developers who have used the languages in practice. The study also tried to find out what are the benefits of using F# compared to C# and if the benefits overcome the extra work that is required to integrate components implemented in another programming paradigm.

This master's thesis is made for Atostek Oy, where C# and F# programming languages are used in software projects. C# is a widely used object-imperative language, and F# is its functional counterpart. Both languages are executed in the Microsoft's .NET framework. In the case of C# and F#, the runtime environment solves most of the problems of integrating software components implemented in different programming languages. However, the problems caused by different paradigms must still be solved.

Results of the interviews show what are the software developers' thoughts on using both languages in practice. The results were consistent and provide a comprehensive view to the practical use of F#.

ALKUSANAT

Tämän diplomityöaiheen on tarjonnut ja rahoittanut Atostek Oy.

Haluan kiittää työn tarkastajaa professori Tommi Mikkosta ja ohjaajaa Juhana Helovuota asiantuntevasta ohjauksesta ja hyvistä kehitysehdotuksista prosessin aikana. Kiitän myös kaikkia Atostek Oy:n työntekijöitä hyvästä ja innostavasta työilmapiiristä. Erityiskiitos kaikille niille henkilöille, jotka kiireiden keskellä löysivät aikaa haastatteluihin osallistumiseen. Ilman teitä työn anti olisi jäänyt paljon vähäisemmäksi.

Kiitän myös isääni ja äitiäni, jotka ovat aina jaksaneet tukea minua niin opinnoissani kuin muillakin elämänaloilla. On helppo keskittyä työhön, kun tietää teidän aina tukevan. Lopuksi vielä kiitos ystävilleni, jotka ovat jaksaneet aikaa vaatineen luovan prosessin aikana kannustaa minua eteen päin. Hiljaa hyvä tulee.

Tampereella 16.4.2014

Tero Lahtinen

SISÄLLYS

1. JOHDANTO

Yksi ensimmäisistä tehtävistä uuden ohjelman toteutuksen alkuvaiheissa on sopivan ohjelmointikielen valinta. Ohjelmoijan on helppo valita kieli, jonka hän jo entuudestaan tuntee. Entuudestaan tuttu kieli ei kuitenkaan aina ole paras vaihtoehto, vaikka uuden kielen opetteluun joutuisikin käyttämään aikaa. Useat tutkimukset kuitenkin osoittavat, että sopivan ohjelmointikielen valinnalla on suuri merkitys ohjelmiston elinkaarikustannuksiin [35]. Ohjelmiston elinkaarikustannuksilla tarkoitetaan kaikkia sen aiheuttamia kustannuksia suunnittelun aloittamisesta aina käytöstä poistoon asti [10, s. 56].

Isoissa järjestelmässä eri osat saattavat olla hyvin erilaisia. Siksi toisinaan saman järjestelmän eri osatkin saatetaan toteuttaa eri kielillä. Tällöin haasteeksi tulee myös eri kielillä toteutettujen osien yhdistäminen. Olemassa on monia hyvin erilaisia ohjelmointikieliä. Jotta ohjelmointikielen valinta voitaisiin tehdä järkevästi, tulee ohjelmistosuunnittelijoilla olla käsitys siitä, millaisia ohjelmointikieliä on olemassa. On tärkeää myös ymmärtää, millaisia haasteita eri kielillä toteutettujen komponenttien yhdistämisessä saattaa olla. Luvussa 2 esitellään ohjelmointiparadigman käsite ja erilaisia paradigmoja. Tässä diplomityössä keskitytään funktionaalisiin ja olio-imperatiivisiin kieliin, joten ne esitellään muita paradigmoja tarkemmin. Tämän lisäksi luvussa tarkastellaan, millaisia haasteita eri kielillä toteutettujen komponenttien yhdistämisessä saattaa olla ja miten yhdistäminen voitaisiin toteuttaa.

Tämä diplomityö on selvittää funktionaalisten ohjelmointikielien vaikutusta käytännön ohjelmistosuunnittelutyöhön. Tarkemman tarkastelun kohteena on Microsoftin .NET-ympäristössä suoritettava F#, jota verrataan samassa ympäristössä yleisemmin käytettyyn C#:iin. Näihin kieliin ja niiden suoritusympäristö .NET:iin tutustutaan luvussa 3. Tämän lisäksi luvussa vertaillaan C#:n ja F#:n ominaisuuksia.

Tämä diplomityö on tehty Atostek Oy:lle (Atostek), jossa on useissa ohjelmistosuunnitteluprojekteissa käytetty funktionaalisia ohjelmointikieliä. Koska olio-imperatiiviset ohjelmointikielet ovat yleisimmin käytettyjä ohjelmointikieliä ja niillä toteutettuja ohjelmia ja komponentteja on olemassa paljon, joudutaan funktionaalisia kieliä käytettäessä varautumaan siihen, että komponenttia käytetään olio-imperatiivisesta ohjelmasta tai että käytetty komponentti on toteutettu olio-imperatiivisella kielellä.

Tässä diplomityössä tutkittiin C#:n ja F#:n käytettävyyttä todellisessa ohjelmis-

toprojekteissa haastattelututkimuksen avulla. Luvussa 4 kuvataan, miten haastattelut tehtiin sekä esitellään haastattelututkimuksen tulokset. Tutkimuksen tavoitteena on vastata seuraaviin tutkimuskysymyksiin:

1. Millainen on F#:n käytettävyys verrattuna C#:iin?
2. Millaisia etuja ja haittoja funktionaalisilla kielillä ja erityisesti F#:lla on verrattuna olio-imperatiivisten ja erityisesti C#:n käyttöön?
3. Millaiseksi ohjelmistosuunnittelijat kokevat F#:lla toteutettujen komponenttien käytön C#:sta?
4. Millaisia tapoja on käärimisen lisäksi tehdä F#:lla toteutettujen komponenttien käytöstä sujuvaa C#:lla?

Luvussa 5 esitellään tämän diplomityön osana toteutettu esimerkkiohjelma, jonka avulla havainnollistetaan C#:n ja F#:n eroja käytännössä. Esimerkkiohjelmassa on toteutettu punamusta puu sekä C#- että F#-kielillä. Luku 6 on yhteenveto työn tuloksista.

2. ERI OHJELMOINTIPARADIGMOJEN HYÖDYNTÄMINEN

Sopivalla kielivalinnalla voidaan pienentää ohjelman elinkaarikustannuksia. Ohjelmointikieliä on kuitenkin hyvin monenlaisia ja ne soveltuvat erilaisten ongelmien ratkaisuun. Ohjelman eri osienkin toteutukseen saattaa olla kannattavaa valita eri ohjelmointikieli ratkaistavan ongelman mukaan. Tässä luvussa käydään läpi, millaisia erilaisia ohjelmointikieliä on olemassa. Lisäksi tarkastellaan mitä haasteita erilaisilla ohjelmointikielillä toteutettujen komponenttien yhdistämisessä saattaa olla ja miten ongelmat voidaan ratakaista.

2.1 Ohjelmointiparadigmat

Kun erilaisia ohjelmointikieliä vertailee, voi huomata, että toiset ovat ominaisuuksiltaan hyvin samankaltaisia ja toiset hyvin erilaisia. Ominaisuuksiltaan samankaltaiset kielet lähestyvät ohjelmointia samankaltaisista lähtökohdista ja samankaltaisilla periaatteilla. Nämä ohjelmoinnin lähtökohdat ja periaatteet muodostavat *ohjelmointiparadigmaksi* (engl. programming paradigm) kutsutun kokonaisuuden.

2.1.1 Ohjelmointiparadigman määritelmä

Ohjelmointiparadigma on yleisesti käytetty termi, mutta sille ei kuitenkaan ole olemassa yhtä yleisesti hyväksyttyä määritelmää. Kaislerin [20, s. 2-3] mukaan Kuhn [21] määrittelee tieteellisen paradigman yleensä käsitteenä, joka kuvaa tieteentekijän tai tutkijan näkemystä maailmasta ja kokonaisuutena teorioista ja oletuksista, jotka vaikuttavat näkemykseen, ja joka on sosiaalisen prosessin tulos. Kuhn käytti paradigman määritelmää fysikaalisen tieteen yhteydessä ja sen hyödyntäminen ohjelmointiparadigmojen yhteydessä on määritelmän alkuperäistä tarkoitusta laajempi [20, s. 3].

Van Roy [41, s. 10] määrittelee ohjelmointiparadigman lähestymistapana ohjelmointiin, joka perustuu matemaattiselle teorialle tai joukolle periaatteita, jotka muodostavat yhtenäisen kokonaisuuden. Harsu [12, s. 15] puolestaan esittää, viitaten Balin ja Grunen teokseen Programming Language Essentials [2], että ohjelmointiparadigma on laskennallisen mallin (engl. model of computation), käsitteistön ja välineistön muodostama kokonaisuus. Laskennallinen malli määrittelee abstraktisti sen,

miten ohjelman suoritus etenee, eli miten tietokone toimii. Käsitteistö muodostuu kielelle ominaisista rakenteista kuten toistorakenteista, osoittimista tai monadeista. Välineistöä on se, miten ohjelmoija käyttää kielen käsitteistöä kuten esimerkiksi tulostusoperaatiot, linkitetty lista tai vaikkapa tilamonadi.

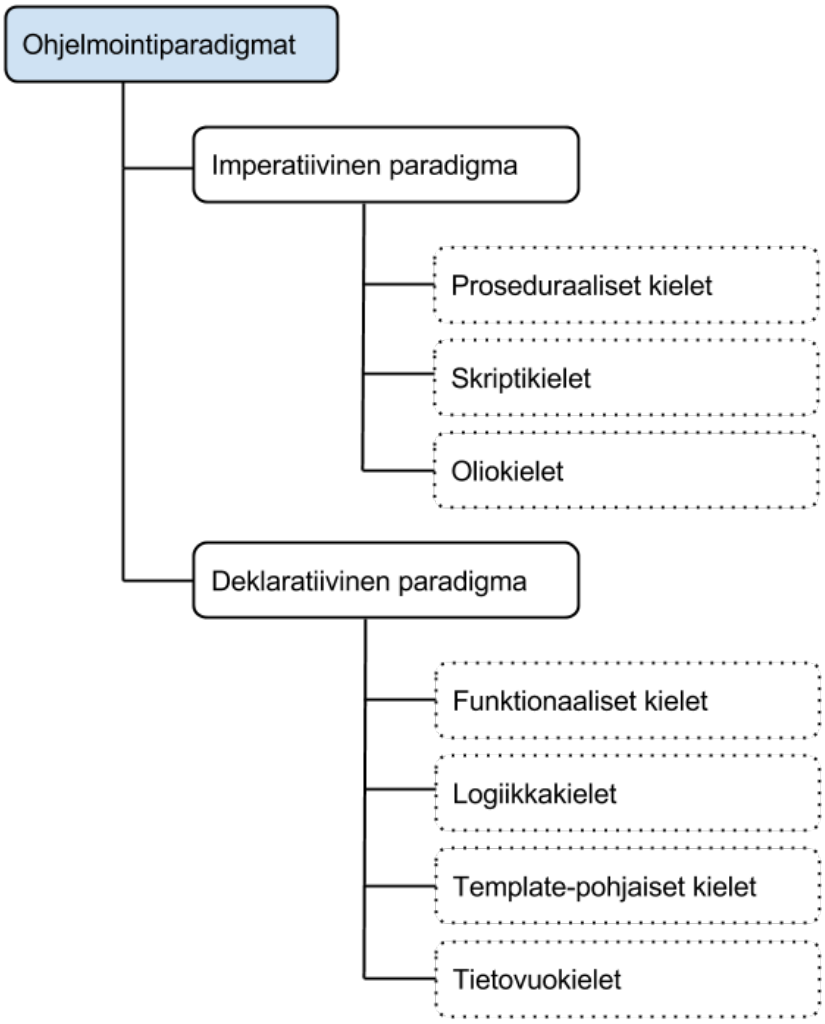
Ohjelmointiparadigma Van Royta, Balia ja Grunea mukaillen on siis *lähestymistapa ohjelmointiin, joka muodostuu laskennallisen mallin, yhtenäisten periaatteiden ja käsitteiden kokonaisuudesta.* Koska samaa paradigmaa noudattavilla ohjelmointikielillä on samat perusperiaatteet, on luonnollista, että näillä kielillä on myös hyvin samankaltaiset välineistöt ja ne ovat siten keskenään hyvin samankaltaisia.

Ohjelmointikielen sanotaan kuuluvan johonkin paradigmaan silloin, kun se tukee paradigman mukaista ohjelmointia. Ei siis riitä, että kieli *mahdollistaa* jonkin paradigman mukaisen ohjelmoinnin, vaan sen on tarjottava luonteva käsitteistö ja välineistö paradigman noudattamiseen. Esimerkiksi C-kielellä voidaan tehdä olio-ohjelmointia, mutta se ei ole olioparadigman kieli, koska se ei tarjoa siihen käsitteistöä. C++-kieli on kehitetty C-kielestä ja on erittäin lähellä sitä. C++-kieli eroaa kuitenkin edeltäjästään siten, että se kuuluu oliokieliin ja tukee olio-ohjelmointia muun muassa perintämekanismilla.

Ohjelmointikieliä voidaan luokitella eri paradigmoihin monin eri tavoin. Jokaisella jaottelutavalla on omat hyvät ja huonot puolensa. Tässä työssä käytetään kuvan 2.1 mukaista jaottelua, joka perustuu Michael Scottin kirjassa Programming Language Pragmatics [45, s. 8-10] esittämään jaotteluun. Siinä kielet on ylimmällä tasolla jaoteltu niiden suunnittelun pohjalla olleen laskennallisen mallin mukaan kahteen ryhmään: *imperatiivisiin* ja *deklaratiivisiin*. Nämä puolestaan voidaan jakaa edelleen osiin kieliin liittyvän käsitteistön ja välineistön pohjalta.

Deklaratiivisille kielille ominaista on, että niillä pyritään kuvaamaan, *mitä tietokoneen pitäisi tehdä*. Niillä siis määritellään laskennan eri osat ja niiden väliset riippuvuudet, mutta ei tarkkaa suoritusjärjestystä. Imperatiivisille kielille puolestaan on ominaista, että niiden laskennallinen malli (tietokone) sisältää aina *muistin*, jonka sisältöä muokkaamalla laskenta etenee ja kielellä määritellään *mitkä operaatiot koneen tulisi suorittaa ja missä järjestyksessä.* Imperatiivisten kielten noudatellessa tietokoneen toimintatapaa muistin käytön osalta, deklaratiivisessa paradigmassa ei sen sijaan ole muokattavan muistin käsitettä ja ovat siten kauempana todellisesta tietokoneen toiminnasta. Siksi voidaankin sanoa, että deklaratiivisten kielien abstraktiotaso on korkeampi kuin imperatiivisten kielten. Deklaratiiviset kielet pyrkivät piilottamaan vähemmän tärkeät toteutusyksityiskohdat ja keskittymään toiminnan ohjaamiseen korkeammalla tasolla. Tästä huolimatta imperatiiviset kielet ovat huomattavasti suositumpia. Niiden suosio on peräisin historiallisesti paremmasta suoritustehokkuudesta ja helpommasta toteutettavuudesta, koska laskentamalli on lähempänä todellista tietokonetta. [45, s. 8]

Scott käyttää proseduraalisista kielistä nimitystä von Neumann -kielet[45, s. 9]. Tämän lisäksi erona hänen käyttämäänsä jaotteluun *skriptikielet* on tässä nostettu proseduraalisten kielten alaryhmästä samalle tasolle oliokielten ja proseduraalisten kielten kanssa. Skriptikielet on nostettu ylemmälle tasolle, koska modernit skriptikielet, esimerkiksi Python, Ruby ja JavaScript, tukevat olio-ohjelmointia erittäin vahvasti ja eivät siten yksiselitteisesti kuuluu proseduraalisten kielten ryhmään. Skriptikielien ominaispiirteet tekevät niistä kuitenkin selkeästi oman kokonaisuuden, jonka vuoksi niitä ei jaeta proseduraallisiin kieliin ja oliokieliin sen mukaan tukevatko ne olio-ohjelmointia vai eivät.



Kuva 2.1: Ohjelmointiparadigmojen luokittelu laskennallisen mallin, käsitteistön ja välineistön perustella. Mukailtu lähteestä [45, s. 9].

Valittu jaottelu ei ole täysin ongelmaton. Se ei esimerkiksi erottele peräkkäistä ja rinnakkaista ohjelmointimallia toisistaan. On olemassa myös muita jaottelutapoja. Esimerkiksi Harsu [12, s. 13-14] esittelee Wegnerin [50] ja Balin ja Grunen [2] käyttämät paradigmajaottelut. Wegnerin käyttämä jaottelu on hyvin samankaltainen tässä

diplomityössä käytetyn jaottelun kanssa. Siinä kielet luokitellaan ylimmällä tasolla deklaratiivisiin ja imperatiivisiin. Deklaratiiviset on luokiteltu edelleen kolmeen alaryhmään: funktionaalisiin, loogisiin ja tietokantakieliin. Imperatiiviset on luokiteltu niin ikään kolmeen alaryhmään: lohkorakenteisiin, oliokeskeisiin ja hajautettuihin (rinnakkaisiin) kieliin. Balin ja Grunen esittelemä paradigmajaottelu erottaa *suoritusmallin* paradigmasta. Näin ohjelmointikielet voidaan luokitella erikseen paradigman (imperatiivinen, oliokeskeinen, funktionaalinen ja looginen) ja suoritusmallin (peräkkäinen, rinnakkainen) mukaan. [12, s. 13-14]

Kummallakin jaottelulla on omat heikkoutensa ja vahvuutensa. Ongelmia on esimerkiksi moniparadigmakielien (engl. multiparadigm language) luokittelussa. Moniparadigmakieli on ohjelmointikieli, joka tukee useampaa paradigmaa. Esimerkiksi F# voidaan tulkita moniparadigmakieleksi, koska se tukee sekä funktionaalista että olio-ohjelmointia. Mikään edellä kuvatuista paradigmojen lajittelutavoista ei ota huomioon moniparadigmakieliä.

Ei ole olemassa mitään yhtä yleisesti hyväksyttyä paradigmoihin luokittelutapaa, vaan luokittelutapa kannattaa aina valita niin, että ohjelmointikielet luokitellaan tutkittavan ongelman kannalta olennaisten ominaisuuksien suhteen. Kuvassa 2.1 esitelty ohjelmointikielien jaottelu onkin tämän diplomityön kannalta riittävä, koska se erottaa selvästi olio-imperatiiviset ja funktionaaliset kielet toisistaan.

2.1.2 Imperatiivinen paradigma

Imperatiivisen paradigman ohjelmointikielillä kirjoitetut ohjelmat voidaan nähdä sarjana komentoja, jotka kertovat tietokoneelle, miten kutakin tietoalkiota pitää muokata, jotta lähtötiedoista saadaan aikaiseksi haluttu lopputulos [52]. Tällainen lähestymistapa on peräisin tietokoneiden von Neumann -arkkitehtuurista. Von Neumann -arkkitehtuurin mukaisissa tietokoneessa on yksi prosessoriyksikkö, joka suorittaa muistipaikkojen sisältöä käsitteleviä käskyjä.[22]

Korkean tason ohjelmointikielten kehitys on saanut alkunsa von Neumann -arkkitehtuurin mukaisten tietokoneiden operaatioiden imitoinnista ja abstrahoinnista. Abstraktiotason nosto helpottaa ohjelmien kirjoittamista ja etenkin ohjelman lukemista ja ymmärtämistä. Lisäksi se helpottaa ohjelmien siirtämistä tietokoneiden välillä, koska laitteistokohtaiset yksityiskohdat on voitu jättää kääntäjän tai tulkin toteutettavaksi.

Toiset imperatiiviset kielet noudattelevat todellisen tietokoneen toimintatapaa vähemmän kuin toiset eli niiden abstraktiotaso on korkeammalla kuin toisten. Ohjelmointiongelmien ratkaisu on helpompaa tehtäväalueen käsitteistöllä kuin koneen käsitteistöllä. Siksi on helpompi ohjelmoida kielillä, joiden käsitteistö on lähempänä sovellusaluetta. Samasta syystä koneenläheisillä ohjelmointikielillä ohjelmointi on vaikeaa ja tuottavuudelta heikompaa, mutta joskus niitä kuitenkin halutaan käyt-

tää teknisistä suorituskykysyistä. Samanlainen toimintaperiaate kuin tietokoneilla on mahdollistanut kielien abstraktiotason nostamisen ja kääntäjien kehittämisen ilman, että ohjelmien tehokkuus on huonontunut. Tämä pätee etenkin niillä imperatiivisilla kielillä, jotka ovat suhteellisen lähellä laitteiston arkkitehtuuria, kuten esimerkiksi C-kieli.

Erilaisista abstraktiotasoista huolimatta kaikille imperatiivisille kielille on kuitenkin yhteistä tapa käsitellä tietoa muuttujien avulla muistipaikkojen arvoja muokkaamalla. Erityisesti sijoituslause on ohjelmointikielten von Neumann -pullonkaula (engl. von Neumann bottleneck), koska se ohjaa ohjelmoijaa ajattelemaan yksityiskohtia kokonaisuuden sijasta. Backus [1] tarkoitti Von Neumann -pullonkaulalla alkuperäisesti von Neuman-arkkitehtuurin mukaisen laitteiston suorittimen ja muistin välisen väylän aiheuttamaa pullonkaulaa. Sen merkitys on kuitenkin laajentunut sen alkuperäisestä merkityksestä käsittämään sekä sijoituslauseen käyttönä näkyvän fyysisen suorittimen ja muistin välisen väylän aiheuttaman pullonkaulan että sijoituslauseen ohjelmointia vaikeuttavan pullonkaulan. Sijoituslause vaikeuttaa ohjelmointia pakottamalla ohjelmoijan päättelemään, mikä muuttujan arvo missäkin tilanteessa voi olla.

Proseduraaliset kielet

Proseduraaliset kielet mukailevat todellisen tietokoneen toimintaa. Näille ohjelmointikielille on ominaista, että muuttujat esittävät muistipaikkoja ja että sijoitusoperaatio sallii ohjelman muokata näiden muistipaikkojen sisältöä. Ohjauskäskyt (engl. control statement, esimerkiksi toisto- ja vaihtoehtorakenteet) korvaavat konekielen vertailu- ja hyppykäskyt [22; 1].

Ohjelma 2.1 on normaali C:n ehtorakenne, ja C-kääntäjä tuottaa siitä ohjelman 2.2 esittämän symbolisen konekielen esityksen. Vertaamalla ohjelmia keskenään on helppo havaita, että ohjelma 2.1 tuottaa saman toiminnallisuuden kuin ohjelma 2.2, mutta C-toteutus on korkeammalla abstraktiotasolla.

Ohjelma 2.1: If-lause C kielellä.

```
bool guard = false;

// ...

if(guard)
{
        // Jos tosi...
}
else
{
        // ...muutoin.
}
```

Ohjelma 2.2: If-lausetta vastaava toiminnallisuus x86-arkkitehtuurin assembly-kielellä.

```
; Tallennetaan ehdolle evaluoitu arvo
; muuttujalle varattuun muistipiakkaan.
        movb    $0, 11(%esp)
        cmpb    $0, 11(%esp)
        je      L2
        ; Jos tosi...
        jmp     L3
L2:
        ; ...muutoin.
L3:
```

Esimerkissä käytetty C on yleisesti käytetty proseduraalinen kieli ja on yksi maailman käytetyimmistä ohjelmointikielistä [3]. Muita proseduraalisia ohjelmointikieliä ovat esimerkiksi Ada-83 [45] ja Fortran, joka on ensimmäinen yleisesti käytetty ohjelmointikieli.

Skriptikielet

Skriptikielet on alunperin tarkoitettu yhdistelemään itsenäisiä erillisiä ohjelmia isommiksi kokonaisuuksiksi, ja ne oli suunniteltu johonkin tiettyyn tarkoitukseen. Esimerkiksi *bash* on komentokieli, jolla voi käynnistää muita ohjelmia ja ohjata niiden suoritusta. JavaScript [34] ja PHP [47] ovat kieliä, joita käytetään pääasiassa dynaamisen sisällön tuottamiseen web-sivuille. JavaScriptiä suoritetaan yleensä selaimessa kun taas PHP-skriptit suoritetaan palvelimella. Toiset skriptikielet, kuten Perl [37], Python [39] ja Ruby [43], ovat puolestaan yleiskäyttöisiä ja hyviä esimerkiksi ohjelmien prototyyppien nopeaan toteuttamiseen. [45]

Skriptikielille on ominaista, että skriptien käyttämiseen ei tarvita erillistä käännösvaihetta, jossa koodista tuotettaisiin suoritettava binääri. Sen sijaan skriptit suoritetaan joko suoraan tulkissa, käännettään ajoaikana juuri ennen suoritusta konekielelle (engl. Just-In-Time compilation, JIT) tai käynnistyksen yhteydessä ne käännetään välikielelle, jota puolestaan ajetaan virtuaalikoneessa. Esimerkiksi JavaScript on skriptikieli, jonka toteutukset ovat yleensä tulkattavia, kuitenkin esimerkiksi Googlen V8 on JIT-kääntäjä. Sen sijaan esimerkiksi Pythonin toteutuksissa skriptit puolestaan käännetään käynnistyksen yhteydessä välikielelle, jota suoritetaan virtuaalikoneessa. [34; 9; 38]

Tulkin ansiosta skriptikielten käyttö on joustavaa. Hyödyntämättä jää kuitenkin kääntäjien mahdollistama käännöksen aikainen virheidentarkistus. Siksi suurin osa virheistä ilmenee ajoaikana. Tämä tekee ohjelmien testaamisesta työläämpää. Skriptikielten toteutukset häviävät myös tehokkuudessa käännettyille kielille, koska tulkin suoritus kuluttaa prosessoriaikaa. Tehottomuuden takia skriptikieliä pidetäänkin usein sopivampina prototyyppien kehittämiseen ja tilanteisiin, joissa vaatimukset

voivat muuttua nopeasti. [16]

Oliokielet

Oliokielet ovat olioparadigman mukaisia ohjelmointikieliä. Olioparadigma voidaan nähdä imperatiivisen paradigman laajennoksena, koska tietoa käsitellään edelleen samalla tavalla kuin proseduraalisissa kielissä. Oliokielille on ominaista, että ohjelmat kootaan reaalimaailman esineitä ja asioita mallintavista olioista. Näin ohjelmoija voi käyttää hyödykseen luonnollista intuitiotaan [22]. Van Roy ja Haridi [42, s. 493] kuvailevat olio-ohjelmointia *ohjelmien luomiseksi kapseloinnin, periytymisen, polymorfismin ja tilallsen suorituksen avulla.* Kapseloinnilla (engl. encapsulation) tarkoitetaan loogisesti yhteen kuuluvan tiedon ja sitä käsittelevän toiminnallisuuden kätkemistä olion sisään niin, että olion käyttäjän ei tarvitse tuntea olion toteutusta. Oliota käytetään sen tarjoaman rajapinnan kautta ja olioista voidaan luoda useita ilmentymiä.

Oliokielet voivat olla joko luokka- tai prototyyppiperustaisia. Luokkaperusteisella oliokielellä tarkoitetaan kieltä, jossa oliot luodaan luokan määrittelyn perusteella ja aliluokat perivät kantaluokkansa rakenteen. Luokkaperustaiselle toimintamallille vaihtoehto on prototyyppiperustainen, joissa oliot luodaan ja periytetään kloonaamalla olemassa oleva olio. Esimerkiksi JavaScript on prototyyppiperustainen oliokieli.

Luokkaperustaista olioparadigmaa tukevat ohjelmointikielet tarjoavat luokaksi (engl. class) kutsutun työkalun olioiden määrittelyyn. Esimerkiksi C++- ja C#-kielissä ja Javassa oliot määritellään luokkien avulla. Olio-ohjelmoinnille olennaista on, että yhden määrittelyn perusteella voidaan luoda useita ilmentymiä (engl. instance).

Luokan määrittelyssä luokan sisäiselle tiedolle ja metodeille eli jäsenmuuttujille ja jäsenmetodeille voidaan määritellä näkyvyysalueita. Erilaisia näkyvyysalueita ovat esimerkiksi yksityinen (engl. private) ja julkinen (engl. public). Yksityiset jäsenmuuttujat ja -metodit ovat vain luokan itsensä käytettävissä eikä niihin pääse käsiksi luokan ulkopuolelta. Julkisiin jäsenmuuttujiin voi kuka tahansa olion tunteva kirjoittaa ja lukea dataa. Vastaavasti julkisia metodeja voi kuka tahansa kutsua. Lisäksi kielet yleensä mahdollistavat erilaisia julkisen ja yksityisen näkyvyyden välimuotoja. Esimerkiksi suojatulla (engl. protected) tarkoitetaan, että muuttujat ja metodit näkyvät alaluokille.

Usein ohjelman osilla (olio-ohjelmoinnin tapauksessa luokilla) on yhteisiä ominaisuuksia eikä samaa toiminnallisuutta haluta monistaa useaan paikkaan. Tämän ongelman ratkaisemiseen on kehitetty periytyminen. Periytyminen on tekniikka, jossa luokka voi uudelleen käyttää toisen luokan toteutuksen, ja usealle luokalle yhteiset toiminnallisuudet voidaan koota yhteiseen kantaluokkaan. Luokka, joka perii, on ali-

luokka ja luokka, jonka ominaisuuksia peritään, on kantaluokka.. Aliluokka voi periä kantaluokan metodien toteutuksen suoraan tai ylikirjoittaa osan niistä.

Oliot piilottavat toteutukseen liittyvät yksityiskohdat siten, ettei olion sisäistä tilaa voida suoraan muokata ulkopuolelta vaan oliota käytetään rajapinnan kautta. Näin olioina toteutettujen kokonaisuuksien väliset suhteet ovat kontrolloituja ja niitä voidaan muuttaa helpommin. Oliot siis ovat työkalu *modulaarisuuden* toteuttamiseen. Ohjelmistosuunnittelussa modulaarisuudella tarkoitetaan ohjelmien jakamista loogisesti erillisiin kokonaisuuksiin ohjelmien monimutkaisuuden hallinnan helpottamiseksi ja toteutettavuuden, ylläpidettävyyden ja ymmärrettävyyden parantamiseksi.

Modulaarisuudella voidaan pyrkiä myös uudelleenkäytettävien komponenttien tekemiseen ja siten työn tuottavuuden parantamiseen [22]. Modulaarisuus ja toteutusyksityiskohtien piilottaminen ei kuitenkaan ole vain olioparadigman erityispiirre, vaan myös muita paradigmoja edustavat kielet usein tarjoavat työkaluja modulaarisuuden tueksi. Esimerkiksi proseduraalisessa Ada83-kielessä pakkauksille (engl. package) voidaan kirjoittaa määrittely (engl. specification) ja toteutus (engl. declaration) erikseen, jolloin pakkauksen käyttäjä ei tiedä toteutusyksityiskohtia [48]. Erona olio-ohjelmointiin pakkauksista ei kuitenkaan voida luoda useita instansseja samaan tapaan kuin luokasta voidaan luoda useita olioita.

Olioparadigman merkitys on kasvanut suureksi viimeisen 20 vuoden aikana, ja sen käytöstä on tullut ohjelmistotuotannon normaali käytäntö [22].

2.1.3 Deklaratiivinen paradigma

Toisin kuin imperatiiviset kielet, joiden perusta on tietokonearkkitehtuureissa, deklaratiiviset kielet perustuvat (abstrakteille) matemaattisille malleille [11]. Esimerkiksi funktionaaliset kielet perustuvat Alonzo Churchin kehittämään lambdakalkyyliin (engl. lambda calculus). Logiikkakielet puolestaan perustuvat predikaattilogiikalle. Matemaattisen perustan ansiosta deklaratiivisilla kielillä tehtyjä ohjelmia on helpompi myös tutkia matemaattisin menetelmin. Esimerkiksi deklaratiivisten ohjelmien ominaisuuksien todistaminen on helpompaa kuin imperatiivisten, koska niiden semantiikka on lähempänä ohjelmien todistamiseen käytettyä predikaattilogiikkaa.

Matemaattisella todistuksella voidaan todeta ohjelman olevan määritelmänsä mukainen. Ohjelman oikea toiminta pitää tällöin määritellä matemaattisen tarkasti; usein loogisina väittäminä. Deklaratiivisten kielien samankaltaisuus predikaattilogiikan kanssa helpottaa ohjelman esittämistä logiikan kaavoina ja vähentää siten ohjelman logiikan lausekkeiksi muuntamisessa tapahtuvia virheitä.

Koska deklaratiivisten kielten perusta on (abstrakteissa) matemaattisissa malleissa eikä fyysisen koneen arkkitehtuurissa, jää myös ohjelmointikielen kääntäjälle

enemmän vapauksia. Kääntäjä voi esimerkiksi pyrkiä rinnakkaistamaan ohjelman suoritusta ilman imperatiivisten kielten rajoituksia. Siksi deklaratiivisista, erityisesti funktionaalisista kielistä on toivottu ratkaisua rinnakkaisten tietokonearkkitehtuurien tehon hyödyntämisessä. Tämän lisäksi niiden on todettu myös nopeuttavan ohjelmistokehitystä. [11]

Funktionaaliset kielet

Funktionaalinen paradigma perustuu Alonzo Churchin 1930-luvulla kehittämälle lambda-kalkyylille. Lambda-kalkyyli kuvaa laskentaa funktioiden avulla. Puhtaasti funktionaalisissa kielissä ei ole sijoituslausetta eikä niissä siksi ole myöskään muuttujia. Puhtaasti funktionaalisissa kielissä funktioilla ei ole sivuvaikutuksia. Funktiokutsu tuottaa tulokseksi vain funktion paluuarvon, jonka arvo riippuu ainoastaan funktiolle annetuiden argumenttien arvoista. [15]

Sivuvaikutusten puuttuminen poistaa myös monia virhelähteitä. Koska funktioilla ei ole sivuvaikutuksia ja sijoituslausetta ei ole, lausekkeiden suoritusjärjestys määräytyy ainoastaan lähtötietojen ja tulosten välisestä suhteesta. Olennaista on, ettei tulosten ja lausekkeiden suoritusjärjestys riipu sivuvaikutusten oikeasta järjestyksestä. Tämän takia funktionaalisten ohjelmien kirjoittaminen ja lukeminen on helpompaa kuin imperatiivisten ohjelmien. Ohjelmoijan ei tarvitse huolehtia ohjelmakoodia kirjoittaessa eikä lukijan lukiessa oikeasta suoritusjärjestyksestä, koska kääntäjä pitää huolen siitä. [15]

Sen lisäksi, että funktionaalisen ohjelman lukeminen on helpompaa, funktionaalisissa kielissä funktioita voidaan käsitellä monipuolisemmin kuin imperatiivisissa kielissä perinteisesti on pystynyt. Funktionaalisissa kielissä funktioita käsitellään datana eli ne ovat niin sanotusti "ensimmäisen luokan kansalaisia". Tämä tarkoittaa, että funktioita voidaan tallentaa muuttujiin (tai nimettyihin arvohin) samaan tapaan kuin muutakin tietoa. Tämän ansiosta funktioita voidaan myös välittää argumentteina muille funktioille ja saada näin aikaan monimutkaisempia funktioita. Funktioita, jotka ottavat argumentikseen toisen funktion tai palauttavat funktion paluuarvonaan, kutsutaan korkeamman asteen funktioiksi (engl. higher order function).

Funktionaalisille kielille on myös ominaista niin kutsuttu kurrisointi (engl. currying) eli funktioiden osittain kutsuminen. Kurrisoinnissa funktiolle annetaan vain osa sen argumenteista. Tällöin tulokseksi saadaan uusi funktio, jolle annetaan argumentiksi vielä sitomatta jääneet argumentit. Yhdessä funktioiden dataluonteen ja kurrisoinnin avulla voidaan luoda uusia erikoistettuja funktioita. Esimerkiksi ohjelmassa 2.3 funktio `double` kertoo annetun arvon kahdella. `List.map` on puolestaan funktio, jolle annetaan kaksi argumenttia: lista ja funktio, jolle argumenttina annetaan listan alkio ja joka tämän perusteella tuottaa uuden arvon. Paluuarvonaan

se palauttaa listan, jossa alkuperäisen listan arvojen perusteella on laskettu uudet arvot. Funktio `doubleAll` on erikoistettu `List.map`-funktiosta niin, että se kertoo kaikki listan alkiot kahdella.

Ohjelma 2.3: Erikoistetun funktion luominen funktioiden dataluonteen ja osittainkutsumisen avulla.

```
let double x = 2 * x
let doubleAll = List.map double

let listOfIntegers = [1; 2; 3; 4; 5]
let doubledValues = doubleAll listOfIntegers
```

Funktionaalisissa kielissä funktioita voidaan yhdistää komposition avulla. Hughes kutsuu artikkelissa Why Functional Programming Matters ohjelmien yhteen liimaamiseksi [15]. Ominaisuus on idealtaan hyvin samankaltainen kuin komentorivikielissä usein käytetty putkitus (engl. pipelining tai piping), jossa ensin suoritetun ohjelman tulosteet ohjataan 'putkea' (engl. pipe) pitkin seuraavalla ohjelmalle.

Funktionaalisille kielille on ominaista, myös lambda-lausekkeiden käyttö. Lambda-lausekkeet ovat nimettömiä funktioita, joilla voidaan esimerkiksi välittää toiminnallisuutta toisille funktiolle. Näin kutsuttavan funktion toiminnallisuutta voidaan muokata tarpeen mukaan. Esimerkiksi ohjelman 2.3 `List.map` funktiolle voitaisiin antaa funktioargumenttina lambda-lauseke, joka kertoo argumenttinsa kahdella. Tällöin yksinkertainen `double`-funktio voitaisiin jättää määrittelemättä ja nimeämättä, jolloin ohjelmasta tulee lyhempi.

Kontrollirakenteet voidaan funktionaalisissa kielissä toteuttaa funktioina ja etenkin toistorakenteet saadaan aikaan rekursiivisina funktioina. Yleisimmin käytetyt kontrollirakenteet ovat myös kirjastoitu. Kirjastoidut kontrollirakenteet säästävät ohjelmoijan samojen rakenteiden kirjoittamiselta yhä uudelleen ja uudelleen. Toisteisen työn lisäksi kirjastoidut kontrollirakenteet vähentävät virheitä, koska ne ovat kattavasti testattuja. Kirjastoidut kontrollirakenteet saadaan toimimaan halutulla tavalla niille argumenttina välitettävän funktion avulla, etenkin lambda-lausekkeita käytetään usein tähän tarkoitukseen. Ohjelman 2.3 `List.map`-funktio on esimerkki tällaisesta kirjastoidusta kontrollirakenteesta.

Osaan funktionaalisista kielistä on toteutettu niin sanottu *laiska laskenta* (engl. lazy evaluation). Laiskassa laskennassa lausekkeita ei evaluoida soveltamisjärjestyksen (engl. application order) perusteella vaan siinä järjestyksessä kun niiden arvoja tarvitaan. Kun lauseke on evaluoitu, sen arvo tallennetaan muistiin ja haetaan sieltä seuraavilla kutsukerroilla. Näin ohjelman suoritusta voidaan tehostaa, koska lopputulokselle tarpeetonta koodia ei koskaan suoriteta ja funktiot suoritetaan tietyillä argumenteilla vain kerran. Laiskan laskennan hyödyntäminen on mahdollista puhtaissa funktionaalissa kielissä, koska muuttuvaa dataa ei ole ja funktioilla ei ole

sivuvaikutuksia, eikä tulokset siten riipu lausekkeiden suoritusjärjestyksestä. Kieliä, joissa on oletusarvona laiska laskenta, kutsutaan laiskoiksi kieliksi. Vastaavasti kieliä, joissa laiskaa laskentaa ei ole, kutsutaan *ahkeriksi*.

Puhtaasti funktionaalisia ohjelmointikieliä ovat esimerkiksi Haskell [13] ja Miranda [40]. Muita yleisesti käytettyjä funktionaalisia kieliä ovat muun muassa Lisp [19] ja F#, jotka sisältävät myös joitain imperatiivisille kielille ominaisia ominaisuuksia. Esimerkiksi F#-kielessä ohjelmoija voi tehdä imperatiivisa toistorakenteita ja luoda olioita. F#:n funktioilla voi myös olla sivuvaikutuksia. Lisäksi kieli on ahkera. Se kuitenkin tukee funktionaalista ohjelmointia erittäin vahvasti. [45]

Logiikkakielet

Logiikkaparadigma pohjautuu matemaattisen predikaattilogiikkaan. Logiikkakielillä kirjoitetut ohjelmat koostuvat loogisista lausekkeista, joista päättelemällä pyritään löytämään kaavoissa esiintyville muuttujille sellaiset arvot, että lausekkeiden tulos on tosi. Koska logiikkakielillä ei kuvata laskennan kulkua vaan ennemminkin haluttu lopputulos, ohjelmakoodi on hyvin lähellä ongelmanmäärittelyä. Koska logiikkakielet perustuvat matemaattiseen logiikkaan, on niiden ominaisuuksien todistaminen suoraviivaisempaa kuin esimerkiksi imperatiivisten ohjelmien. [12]

Prolog on esimerkki yleiskäyttöisestä logiikkakielestä. Sitä on hyödynnetty esimerkiksi asiantuntijajärjestelmissä (engl. expert system), luonnollisten kielten käsittelyssä (engl. natural language processing), peleissä ja muissa tekoälyyn liittyvissä sovelluskohteissa. Prologille on olemassa sekä tulkkeja että kääntäjiä ja niistä on olemassa useita eri toteutuksia, esimerkiksi GNU Prolog [6] ja SWI-Prolog [46]. Prologista on olemassa toteutus myös Microsoftin .NET-ympäristöön, Prolog.NET [14].

2.2 Haasteet eri ohjelmointikielillä toteutettujen ohjelmakomponenttien yhdistämisessä

Eri ohjelmointikielillä toteutettujen komponenttien yhdistäminen saattaa olla monimutkaista. Ongelmia voivat tuottaa esimerkiksi kielien erilaiset *tyyppijärjestelmät*. Tyyppijärjestelmän avulla ohjelmointikielissä määritellään, miten tietokoneen muistissa olevat bittijonot tulee tulkita. Se luokittelee bittijonot erilaisiksi tyypeiksi tai arvojoukoiksi, jonka arvot tulkitaan samalla tavalla. Tyyppijärjestelmä määrittelee myös, mitä eri tyypeillä voi tehdä ja miten tyypit toimivat yhdessä. Esimerkiksi muistipaikan sisältämä bittijono voidaan tulkita tyypistä riippuen joko kokonaisluvuksi tai osoittimeksi toiseen muistipaikkaan.

Tyyppijärjestelmän avulla tehdä *tyyppitarkastuksia* ja varmistaa, että arvoja käsitellään tyyppijärjestelmän sääntöjen mukaisesti. Tyyppitarkastukset auttavat löy-

tämään virheitä ohjelmista ja lisäävät siten ohjelmien toimivuutta ja turvallisuutta. Tyyppitarkastelua voidaan tehdä ohjelman ajon aikana, jolloin kyseessä on dynaaminen tyyppitarkastus. Tyyppivirheen löytyessä myös virhe pitää käsitellä ajon aikana. Mikäli ohjelmassa ei ole varauduttu virheeseen, tyypillisesti ohjelman suoritus keskeytetään. Tarkastukset voidaan tehdä myös käännösaikana, tällöin puhutaan staattisesta tyyppitarkastuksesta. Staattisen tyyppitarkastuksen etuna on, että virheet tulevat esille aikaisemmin, jolloin niiden korjaaminen on yleensä nopeampaa ja ylipäätään mahdollista ennen kuin ohjelmaa suoritetaan. Dynaamisessa tyyppitarkastuksessa ohjelmaa pitää suorittaa kunnes tyyppivirhe tulee esille.

Tyyppijärjestelmien erot aiheuttavat aina ongelmia, kun eri ohjelmointikielillä toteutettuja ohjelmanosia yhdistetään. Tämä ongelma ei ole vain paradigmojen välinen, vaan myös samaa paradigmaa edustavien kielien tyyppijärjestelmät eroavat toisistaan. Tyyppijärjestelmien yhteensovittamisesta tulee sitä vaikeampaa mitä enemmän kielet ja niiden perusperiaatteet eroavat toisistaan, koska myös paradigmoille tyypilliset tietorakenteet vaihtelevat. Ohjelmoijan tulee ottaa huomioon, miten ohjelmointikielen tietotyypit esitetään toisessa kielessä [44]. Esimerkiksi C++-kieli eroaa edeltäjästään C-kielestä jopa perustietotyypeiltään. C-kielessä ei ole C++-kielestä tuttua totuusarvoa edustavaa perustietotyyppiä `bool`. Totuusarvoinen tietotyyppi on tuotu C-kieleen vasta vuonna 1999, jossa se saadaan käyttöön ottamalla käyttöön standardikirjaston `stdbool.h`-esittelyt.

Vaikka ohjelmointikielissä olisi vastaavat tietotyypit, niiden toteutusyksityiskohdat voivat kuitenkin erota toisistaan. Ohjelmoijan on varmistettava, että tietotyypit ymmärretään samoin ja että niitä käsitellään oikein eri kielillä toteutetuissa komponenteissa [44]. Esimerkiksi C++-kielessä kokonaislukutietotyypin (`int`) koko vaihtelee sen mukaan, mille laitteistolle ohjelma on käännetty. Vastakohtana tälle on esimerkiksi C#, jossa kokonaislukutietotyyppi (`int`) on aina 32:n bitin mittainen. Mikäli C#-kielellä toteutetusta ohjelmanosasta tuodaan kokonaislukumuuttuja C++:llä toteutettuun ohjelmanosaan, pitää kokoero ottaa huomioon. Mikäli C++-osa on käännetty 32-bittiselle arkkitehtuurille, kokonaislukumuuttuja on saman kokoinen kuin C#:n kokonaislukumuuttuja. Tilanne on toinen, mikäli C++-osa on käännetty 16-bittiselle arkkitehtuurille. Tällöin C#-kielen kokonaislukumuuttujan tallentaminen suoraan `int`-tyyppiseen muuttujaan voi aiheuttaa ylivuodon ja luvun suuruus saattaa muuttua.

Koon lisäksi myös perustietotyyppien toteutustavat voivat erota toisistaan. Esimerkiksi Javan `int`-tietotyyppiä C#-kielessä vastaa sen kokonaislukutyyppi `int`. Siinä missä Javan kokonaislukutietotyyppi edustaa vain kokonaisluvun arvoa, C#-kielessä kokonaislukutietotyyppi on olio, jolla on omat metodit. Esimerkiksi `ToString()`-metodilla voidaan C#-kielessä kokonaisluku muuttaa arvoa esittäväksi merkkijonoksi (`string`).

Voi olla, että komponentti on toteutettu kielellä, jossa on käsitteitä, joita ei komponenttia käyttävästä kielestä löydy. Esimerkiksi jos C++-kielellä toteutettu komponentti palauttaa osoittimen olioon, ei C-kielisestä ohjelmasta käytettynä ohjelmoijalla ole käytössään valmiita työkaluja olion käsittelyyn. Olion jäsenmuuttujien arvoja lukeakseen ohjelmoijan on tunnettava kääntäjän generoima olion rakenne ja olion käsittelystä tulee erittäin virhealtista. Myöskään olion metodeiden dynaaminen sitominen (engl. dynamic binding) ei toimi, vaan käyttäjän on tiedettävä, mitä versiota funktiosta pitää kutsua milloinkin.

2.3 Eri ohjelmointikielillä toteutettujen komponenttien yhdistäminen

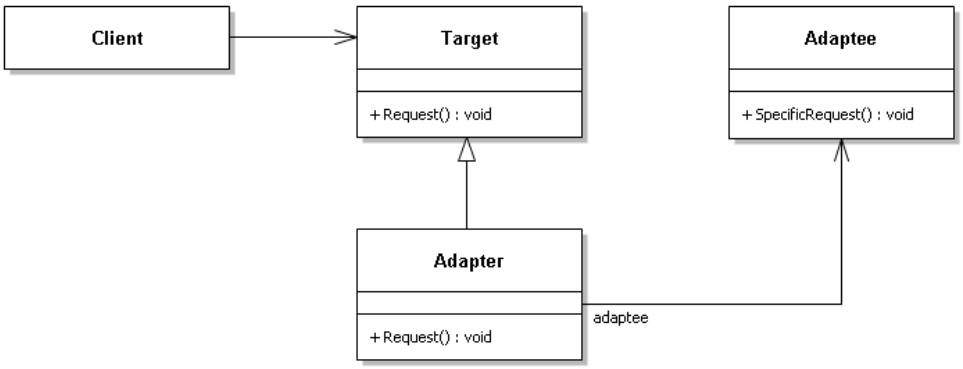
Ohjelman osat voidaan yhdistää monella eri tavalla. Järjestelmän osat voidaan yhdistää esimerkiksi integrointialustalla (engl. enterprise integration environment). Tällöin osat on hajautettu ja ne ovat yhteydessä tietoverkon yli. Verkko erottaa osat toisistaan luonnollisesti, jolloin ohjelmoijat voivat tuottaa eri osat eri kielillä riippumatta muiden osien kielistä. [49] Osat kommunikoivat keskenään esimerkiksi XML- tai JSON-viestien välityksellä. Tällöin viestin lähettäjän pitää luoda ja vastaanottajan tulkita viestien sisältö oikein. Osien toteutuksissa käytetyt ohjelmointikielet eivät kuitenkaan vaikuta datan tulkintaan, vaan ainoastaan viestiprotokolla.

Ilman verkkoa eri kielillä toteutettujen komponenttien integroinnissa virtuaalikoneet voivat olla hyödyllisiä. Esimerkiksi Microsoftin Common Language Runtime (CLR) tukee monia erilaisia ohjelmointikieli kuten esimerkiksi C#, Visual Basic ja F#. Vastaavasti Javan virtuaalikone (engl. Java virtual machine, JVM) tukee monia erilaisia kieliä, kuten esimerkiksi Java, Scala ja Jython (Pythonin JVM-toteutus). Molemmissa mainituissa virtuaalikoneissa on oma tyyppijärjestelmä, jota niille käännettävät kielet käyttävät. Tämä poistaa monta perustietotyyppien yhteensovittamisen ongelmaa. Ratkaisematta jäävät edelleen paradigmojen epäyhteensopivuudet. [49]

Eri kielien ja paradigmojen erot voidaan pyrkiä piilottamaan hyödyntämällä kääre-suunnittelumallia (engl. wrapper tai adapter). Kääre on suunnittelumalli, jolla muunnetaan jonkin kohdeolion rajapinta asiakasolion ymmärtämään muotoon. Kääreen avulla sellaiset oliot, joilla muuten olisi epäyhteensopivat rajapinnat, voivat kommunikoida keskenään. Kääreen rakenteen periaate on esitetty kuvassa 2.2. [8, s. 139 - 150]

Eri kielellä toteutettu valmiskomponentti voidaan siis kääriä niin, että komponenttia käyttävässä koodissa ei tarvitse huomioida komponentin todellista toteutuskieltä. Esimerkiksi tilanteessa, jossa ohjelma ollaan toteutettu kielellä L1, mutta valmiskomponentti on toteutettu kielellä L2, voidaan valmiskomponentille toteuttaa

kääre L1-kielellä ja piilottaa näin eri ohjelmointikielen käyttö.



Kuva 2.2: Sovitin -suunnittelumalli. Mukailtu lähteestä [8].

Käärimisen etuna on, ettei valmiskomponentin lähdekoodeja tarvita. Mikäli käytettävät kielet käännetään jollekin virtuaalikoneelle, kääreiden toteuttamiseen vaadittava työmäärä pienenee huomattavasti perustietotyyppien pysyessä samoina. Kääreellä voidaan piilottaa helposti myös paradigmojen eroavaisuudet. Esimerkiksi funktionaalisella kielellä toteutettu komponentti ei voi muuttaa tilaansa vaan sen pitää paluuarvona palauttaa itsestään uusi versio. Imperatiivisella kielellä ohjelmoivan ohjelmoijan mielestä tällainen toiminnallisuus voi olla vaikeasti ymmärrettävä ja hankalasti käytettävä. Tällöin myös riski, että ohjelmoija unohtaa ottaa talteen komponentin uuden version, kasvaa. Kääreellä tämä toiminnallisuus voidaan piilottaa niin, että imperatiivinen kääre tallentaa aina uusimman version kääritystä komponentista.

3. FUNKTIONAALISTEN KIELTEN HYÖDYNTÄMINEN KÄYTÄNNÖSSÄ

Tämä luku esittelee .NET-ympäristön ja miten se vaikuttaa eri ohjelmointikielillä toteutettujen komponenttien yhdistämiseen. Lisäksi esitellään F#- ja C# -ohjelmointikielet ja vertaillaan niiden ominaisuuksia. Lopuksi esitellään, miten Atostek Oy:ssa on hyödynnetty F#:ia ja muita funktionaalisia ohjelmointikieliä ohjelmistosuunnitteluprojekteissa.

3.1 .NET Framework

Microsoftin kehittämä .NET Framework (.NET) on ohjelmistoalusta, jota käytetään lähinnä Microsoftin erilaisissa Windows-ympäristöissä. Se tarjoaa oliopohjaisen ohjelmien kehitys- ja suoritusympäristön, jolla pyritään tehostamaan ohjelmien tekemistä ja vähentämään versioristiriitoja. Se koostuu ajoympäristöstä (engl. Common Language Runtime, CLR) ja luokkakirjastosta. Ajoympäristön ansiosta kolmansien osapuolien kehittämien ohjelmistokomponenttien suorittaminen on turvallisempaa, koska se antaa niille oikeuksia suorittaa erilaisia operaatioita esimerkiksi komponenttien alkuperän perusteella. [26]

.NET on ohjelmointikieliriippumaton, ja sille kehitetyt ohjelmat käännetään välikielelle (engl. Common Intermediate Language, CIL), jota suoritetaan ajoympäristön virtuaalikoneessa. Ohjelmoija voi siis valita ongelmaan parhaiten sopivan kielen komponentin toteutuskieleksi ja voi edelleen käyttää kaikkia .NET-ympäristön kirjastoja välittämättä siitä, millä kielellä ne on toteutettu. Kielen valinnassa ainoa vaatimus on, että kielelle on olemassa kääntäjä CIL-välikielelle. .NET-ympäristössä käytettäviä ohjelmointikieliä ovat esimerkiksi C#, F#, C++, IronPython [17] (Python kielen toteutus .NET-ympäristöön), IronRuby [18] (Ruby kielen toteutus .NET-ympäristöön) ja Visual Basic. [25]

.NETin ajoympäristö huolehtii välikielisen koodin suorituksesta ja muistin- ja säikeidenhallinnasta. Ajoympäristö huolehtii niin kutsusta JIT-kääntämisestä (Just In Time), jonka ansiosta ohjelmaa ei koskaan tulkata vaan välikieli käännetään ajoaikaisesti konekielelle. Ajoympäristö myös valvoo ohjelmien oikeuksia. Esimerkiksi ohjelman oikeus lukea ja kirjoittaa tiedostoihin riippuu sen alkuperästä, eli onko se lähtöisin internetistä, yritysverkosta vai paikalliselta koneelta. [26]

.NET-ajoympäristöön kuuluu myös yhteinen tyyppijärjestelmä (engl. Common

Type System, CTS), joka määrittelee miten tyyppejä voidaan määritellä ja käyttää sekä miten niitä hallitaan ajoaikana. CTS:ään kuuluu kahdenlaisia tyyppejä: arvotyyppejä (engl. value type) ja viitetyyppejä (reference type). Muuttujat, joiden tyyppi on arvotyyppiä, on sidottu suoraan niihin sijoitettuun arvoon, viitetyyppiset muuttujat puolestaan ovat ainoastaan osoittimia varsinaisiin arvoihin. Arvotyyppejä ovat esimerkiksi CTS:n tarjoamat perustietotyypit kuten boolean, byte, char ja 32 bittinen int, myös ohjelmoijan määrittelemät tietueet (engl. structure) ja enumeraatiot (engl. enumeration) ovat arvotyyppejä. Viitetyyppejä ovat esimerkiksi luokat ja delegaatit (engl. delegate). [23]

CTS:n ansiosta voidaan eri ohjelmointikielillä kirjoitettuja komponentteja yhdistää, koska ne käännetään samalle tyyppijärjestelmälle ja CTS määrittelee tyyppisäännöt, joita kielien on noudatettava [23]. Näin vältytään monilta kielten tyyppijärjestelmien ristiriidoilta ja epäyhteensopivuuksilta etenkin perustietotyyppien osalta. CTS ei kuitenkaan ratkaise kaikkia ongelmia, koska eri ohjelmointikielillä määritellyt tietotyypit voivat olla vaikeita käyttää toisista ohjelmointikielistä. Ohjelmointikielten ja etenkin paradigmojen erot korostuvat erityisesti monimutkaisten tietotyyppien kohdalla.

3.2 Olio-imperatiivinen C#

C# on Microsoftin kehittämä yleiskäyttöinen ohjelmointikieli, jota käytetään lähinnä .NET-ympäristössä. Kielen ensimmäinen versio julkaistiin vuonna 2000, ja kielelle myönnettiin ECMA -standardi vuonna 2002. C#-kielen pääkehittäjiä ovat olleet Aders Hejlsberg, Scott Wiltamuth ja Peter Golde. [7]

C# on yleiskäyttöinen luokkaperustainen olio-imperatiivinen kieli, jonka suunnittelutavoitteena on ollut yksinkertaisuus. Sen on tarkoitus olla helposti opittava etenkin niille ohjelmoijille, jotka tuntevat joko C- tai C++-kielen entuudestaan. C#-kielen tavoitteena on tehostaa ohjelmistotuotantoa ja vähentää virheitä C++ -ohjelmointiin verrattuna tarjoamalla

- vahvan käännösaikaisen tyyppitarkastuksen,
- käännösaikaisen alustamattomien muuttujien käytön tarkastuksen,
- ajoaikaisen taulukoiden rajojen tarkastuksen ja
- automaattisen roskienkeruun. [7]

C# on suunniteltu sopivaksi hyvin erilaisiin ohjelmistoihin aina pienistä, tiettyihin toimintoihin suunnitelluista sulautetuista ohjelmistoista suuriin ja monimutkaisiin järjestelmiin. Se on suunniteltu tehokkaaksi sekä muistin että suorituskyvyn kannalta, mutta sen ei ole kuitenkaan tarkoitus kilpailla C-kielen ja assembly-kielien kanssa suorituskyvyssä eikä ohjelmien koossa [7]. Käytännössä C# ei kuitenkaan

sovellu aivan yksinkertaisimmille laitteistoille, koska sitä käytetään lähinnä .NET-ympäristössä. Suoritusympäristön tulee siis kyetä suorittamaan myös CLR:ää ja Windows-käyttöjärjestelmää.

C#-kielessä on vahva staattinen tyypitys, jonka perustana on .NET-ympäristön CTS. C#-kielessä tyypit ovat siis joko arvotyyppejä tai viitetyyppejä. Arvotyyppeihin kuuluvat perustietotyypit (kuten `char`, `int`, `float` ja `double`), `enum`- ja `struct`-tyypit. Viitetyyppejä ovat luokat, rajapintatyypit, delegaattityypit ja taulukkotyypit. C#-kielessä kaikki tyypit periytyvät `object`-tyypistä. Myös perustietotyypit ovat siis olioita ja periytyvät `object`-tyypistä ja esimerkiksi kokonaislukuarvoille voidaan suorittaa metodikutsuja, kuten ohjelmassa 3.1 on tehty.

Ohjelma 3.1: C#-kielessä myös perustietotyyppien arvot ovat olioita. Siksi niille voidaan tehdä metodikutsuja. Alla oleva ohjelma tulostaa merkkijonon "42".

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(42.ToString());
    }
}
```

Nimiavaruudet (engl. namespace) ovat C#-kielen työkalu ohjelman osien organisointiin niin ohjelmassa sisäisesti kuin ulkopuolisellekin käyttäjälle. C#-kielessä nimiavaruudet voivat sisältää myös alinimiavaruuksia kuten ohjelmassa 3.2 on havainnollistettu. Nimiavaruuksia voi määritellä myös osissa eli samaa nimiavaruutta voidaan määritellä esimerkiksi useammassa tiedostossa. C#:n nimiavaruudet toimivat samoin kuin .NET-ympäristön nimiavaruudet. [33]

Ohjelma 3.2: C#-kielessä nimiavaruudet voivat sisältää myös alinimiavaruuksien määrittelyjä. Alla on esitetty kaksi eri tapaa esitellä alinimiavaruus. Molemmat tuottavat saman lopputuloksen.

```
namespace Namespace
{
   namespace NestedNamespace
   {
      // Code here...
   }
}

namespace Namespace.NestedNamespace
{
   // Code here...
}
```

Kuten nimiavaruudet, myös luokat voidaan C#-kielessä määritellä osissa. Luokan määrittelyä kahdessa osassa on havainnollistettu ohjelmassa 3.3.

Ohjelma 3.3: C#-kielessä myös luokat voidaan määritellä osittain. Alla olevassa ohjelmassa luokan määrittely on jaettu kahteen osaan.

```
public partial class ExampleClass
{
    // The first part of a declaration here...
}

public partial class ExampleClass
{
    // The second part of the declaration here...
}
```

Tapahtumien (engl. event) avulla luokat ja oliot voivat ilmoittaa muille luokille ja olioille erilaisista tapahtumista. Tyypillisesti niitä käytetään esimerkiksi käyttöliittymässä käyttäjän toimista ilmoittamiseen. Tapahtuman lähde päättää, milloin tapahtuma laukeaa ja millä parametreilla. Tapahtuman käsittelijä päättää, miten tapahtuma käsitellään. Yhdellä tapahtumalla voi olla useita käsittelijöitä. [32] C#:n tapahtumat perustuvat .NET-ympäristön tapahtumamekanismille.

Delegaatti (engl. delegate) on tyyppi, joka määrittelee metodin tyypin (engl. signature). Delegaatti-instanssiin voidaan sitoa määritellyn tyypin mukaisia metodeja, jolloin niitä voidaan kutsua kutsumalla delegaattia, joka pitää huolen sidottujen metodien kutsumisesta. Tämän ansiosta kutsuttavaa metodia ei tarvitse tietää vielä käännösaikana vaan sitominen voidaan tehdä dynaamisesti. Dynaamisuudesta huolimatta delegaatin tyyppimäärittelyn ansiosta voidaan kuitenkin olla varmoja, että dynaamisesti sidottua metodia voidaan kutsua halutuilla parametreilla ja paluuarvo on haluttua tyyppiä. Esimerkiksi C#-kielen tapahtumamekanismi hyödyntää delegaatteja tapahtumankäsittelijöitä kutsuttaessa. C#:n delegaatit perustuvat .NET-ympäristön delegaattimekanismille. [24]

C#-kielellä ei ole omaa luokkakirjastoa, vaan se hyödyntää .NET-ympäristön luokkakirjastoa. Tästä hyvänä esimerkkinä on Language-Integrated Query-komponentti (LINQ), joka mahdollistaa selkeiden deklaratiivisten kyselyiden kirjoittamisen. LINQun avulla voidaan suorittaa kyselyjä esimerkiksi tietokantoihin, XML-dokumentteihin ja .NET-luokkakirjaston tietorakenteisiin. Ohjelmassa 3.4 on esitetty yksinkertainen esimerkki, jossa taulukosta suodatetaan ne alkiot, joiden arvo on alle 10, kerrotaan ne kahdella ja tulostetaan. Vertailukohtana voi käyttää ohjelmaa 3.5, jossa LINQ-kysely on korvattu `foreach`-silmukalla.

C# tukee myös reflektiota (engl. reflection). Reflektion avulla ohjelmassa voidaan tarkastella ajonaikaisesti olioiden tyyppi-informaatiota kuten tyypin nimeä ja metodien nimiä ja argumentteja. Sen avulla voidaan luoda olioita ja päättää luotavan olion tyyppi ajoaikaisesti. [27]

.NET-ympäristöön suunniteltuja C#-kielisiä ohjelmia ei käännetä konekielel-

Ohjelma 3.4: Ohjelma, jossa taulukosta suodatetaan kaikki 10 pienemmät kokonaisluvut LINQ-kyselyllä ja tulostetaan ne.

```
int[] numbers = new int[] { 1, 3, 6, 11, 42, 5, 67 };

IEnumerable<int> smallNumbers =
   from int n in numbers
   where n < 10
   select 2 * n;

foreach (int n in smallNumbers)
{
    Console.WriteLine(n.ToString());
}
```

Ohjelma 3.5: Ohjelma on toiminnaltaan sama kuin ohjelma 3.4, mutta LINQ-kysely on korvattu perinteisellä `foreach`-silmukalla.

```
int[] numbers = new int[] { 1, 3, 6, 11, 42, 5, 67 };

List<int> smallNumbers = new List<int>();

foreach (int n in numbers)
{
    if (n < 10)
    {
        smallNumbers.Add(2 * n);
    }
}

foreach (int n in smallNumbers)
{
    Console.WriteLine(n.ToString());
}
```

le vaan .NET-ympäristöön kuuluva C#-kääntäjä kääntää lähdekoodin .NET-ympäristön virtuaalikoneessa suoritettavaksi CIL-välikoodiksi. Kieli itsessään ei ole .NET-sidonnainen ja kielelle voitaisiinkin toteuttaa kääntäjiä myös muihin ympäristöihin. Koska C#:lla ei ole omia standardikirjastoja, muissa ympäristöissä ohjelmoijan tulisi kuitenkin toteuttaa itse kaikki sellainen toiminnallisuudet joissa .NET-ympäristössä tukeudutaan ympäristön valmiisiin komponentteihin. Tällaisiin ominaisuuksiin kuuluvat esimerkiksi yleisesti käytetyt tietorakenteet lista ja sanakirja (engl. dictionary).

3.3 Funktionaalinen F#

F# on Microsoftin Don Symen johdolla kehittämä ohjelmointikieli, jonka pääparadigma on funktionaalisuus, ja se on lähellä ML-kieltä [5]. Funktionaalisuuden lisäksi F# tukee myös proseduraalista ja olioparadigman mukaista ohjelmointityyliä. F#-kieltä käytetään .NET-ympäristössä ja se käännetään .NET-virtuaalikoneen välikielelle [28]. Välikielikäännöksen ansiosta myös muilla .NET-ympäristön kielillä voidaan käyttää F#-kielellä toteutettuja komponentteja ja F#:lla toteutetuissa ohjelmissa voidaan hyödyntää muilla kielillä toteutettuja komponentteja.

F#-kielen funktiot ovat arvoja. Tämän ansiosta niitä voidaan sitoa muuttujiin ja esimerkiksi välittää argumentteina toisille funktioille. F# mahdollistaa myös *funktioiden komposition*. Kompositiossa muodostetaan kahden funktion yhdistelmänä uusi funktio.

Tyyppipäättely on merkittävä osa F#-kieltä. Tyyppipäättelyn ansiosta lausekkeiden tyyppiä ei tarvitse kertoa tilanteissa, joissa tyyppipäättelijä osaa ohjelmakoodin perusteella päätellä tyypit itse. Tämä selkeyttää ohjelmakoodia, koska usein lausekkeiden tyypit ovat asiayhteyden perusteella ohjelmoijalle itsestään selviä, eivätkä tyyppimäärittelyt siten vie ohjelmoijan huomiota olennaisesta. Tyyppipäättelyn huonona puolena on, että ohjelmoija saattaa jättää tyyppimäärittelyt kirjoittamatta myös tilanteessa, joissa ohjelmaa tuntemattomalle tyypit eivät ole itsestään selviä. Tyyppipäättelyn vaikutusta ohjelmakoodiin on havainnollistettu ohjelmissa 3.6 ja 3.7. Ohjelmassa 3.6 kaikkien funktion argumenttien, funktion paluuarvon ja arvojen tyypit on määritelty, ohjelmassa 3.7 tyypit on puolestaan jätetty merkitsemättä.

F#:ssa on mahdollista määritellä vaihtoehtotyyppejä (engl. discriminated union). Ne ovat tietotyyppejä, jotka määrittelevät joukon nimettyjä valintoja. Eri valinnoille voidaan myös määritellä sisällöksi eri tyypit. Niille ei kuitenkaan välttämättä tarvitse määritellä sisältöä. Tällöin vaihtoehtotyyppi vastaa `enum`-tietotyyppiä. Ohjelmassa 3.8 on havainnollistettu vaihtoehtotyyppiä, jota voitaisiin käyttää erilaisten muotojen piirtotietojen tallentamiseen.

F# -kielen ohjelmoinnille on ominaista `Option`-tyyppin käyttö. `Option` on genee-

Ohjelma 3.6: F#-kielen funktiomäärittely, jossa funktion paluuarvon, argumenttien ja arvojen tyypit on määritelty eksplisiittisesti. Funktio saa kutsuparametreinaan kaksi `string`-arvoa ja yhden `int`-arvon. Funktio yhdistää kaksi ensimmäistä argumenttia, kertoo kolmannen argumentin kahdella ja palauttaa tulokset parina.

```
let combineStringsAndMultiplyBy2
   (arg1 : string)
   (arg2 : string)
   (arg3 : int) : (string * int) =
      let combinedString : string = arg1 + arg2
      let multipliedInt : int = 2 * arg3
      (combinedString, multipliedInt)
```

Ohjelma 3.7: F#-kielen funktiomäärittely, jossa funktion paluuarvon, argumenttien ja arvojen tyypit on jätetty tyyppipäättelijän pääteltäväksi. Funktio on toiminnallisuudeltaan sama kuin ohjelma 3.6.

```
let combineStringsAndMultiplyBy2 arg1 arg2 arg3 =
   let combinedString = arg1 + arg2
   let multipliedInt = 2 * arg3
   (combinedString, multipliedInt)
```

Ohjelma 3.8: Ohjelmassa luodaan vaihtoehtotyyppi Shape, joka kuvaa geometristen muotojen piirtoarvoja. Vaihtoehtotyypin määrittelyn alapuolella luodaan nelikulmio.

```
type Shape =
   | Rectangle of width : float * length : float
   | Circle of radius : float
   | Line of length : float

let square = Rectangle(width = 4.2, length = 4.2)
```

rinen tietotyyppi, jolla voi olla joko arvo `None` tai `Some`. Sen avulla voidaan välttää `null`-arvojen käyttäminen (vertaa esimerkiksi Java). Puuttuva tai "tyhjä" arvo voidaan tällöin ilmaista `None`-arvolla. Arvolle `Some` annetaan haluttu data hyötykuormaksi. `Option`-rakenteen määrittely on esitetty ohjelmassa 3.9.

Tyypillisesti esimerkiksi funktiot voivat palauttaa `Option`-tyyppisen olion, jolloin funktion kutsujan pitää tehdä tarkastelu, onko paluuarvo `None` vai `Some`. Mikäli paluuarvo on tyyppiä `Some`, voidaan varsinainen paluuarvo kääriä ulos rakenteesta. Menettely vähentää huomattavasti poikkeuksien, tyypillisesti `NullReferenceException`, määrää, koska ohjelmassa ei käsitellä suoraan `null`-arvoja.

Ohjelma 3.9: Option-tietotyypin määritelmä.

```
type Option<'a> =
    | Some of 'a
    | None
```

F#-kielen käytölle on ominaista `match`-valintarakenteen käyttö. Niitä käytetään etenkin vaihtoehtotyyppien arvojen tarkasteluun. Vaihtoehtotyypit onkin optimoitu juuri `match`-valintarakenteen kanssa käytettäväksi [28]. F#:n kääntäjä pakottaa ohjelmoijan ottamaan huomioon kaikki tietotyypissä mahdolliset vaihtoehdot. Tämä vähentää ohjelmaan jäävien virheiden määrää. Ohjelmassa 3.10 on havainnollistettu `match`-rakenteen käyttöä.

Ohjelma 3.10: Ohjelma havainnollistaa valintarakenteen käyttöä ohjelman 3.8 Shape-vaihtoehtotyypin kanssa.

```
let square = Rectangle(width = 4.2, length = 4.2)
match square with
| Rectangle(width,length) when width = length ->
    Console.WriteLine("It's␣square!")
| Rectangle _ ->
    Console.WriteLine("It's␣a␣rectangle!")
| Circle r ->
    Console.WriteLine("Circle's␣radius␣is␣{0}!", r)
| _ ->
    // The only possible value at this point is Line,
    // but for example it's ignored here.
    Console.WriteLine("It's␣something␣else!")
```

F#-kielelle kuten muillekin funktionaalisille kielille on tyypillistä lambda-lausekkeiden käyttö. Lambda-lausekkeet ovat nimeämättömiä funktioita. Niitä voidaan käyttää esimerkiksi tilanteissa, joissa tarvitaan hetkellisesti toistettavaa toiminnallisuutta niin, ettei sitä tarvitse nimetä. F#-kielessä niitä käytetään tyypillisesti

esimerkiksi kirjastoitujen toistorakenteiden yhteydessä. Ohjelma 3.11 havainnollistaa lambda-lausekkeiden käyttöä listojen käsittelyssä. Ensin ohjelmassa luodaan lista, jossa on kokonaisluvut yhdestä sataan. Seuraavassa kohdassa listasta suodatetaan pois parittomat luvut, kerrotaan jäljelle jääneet luvut kahdella ja summataan ne. Toiminnallisuus voitaisiin suorittaa yksinkertaisemminkin kuin esimerkissä on tehty. Esimerkin tarkoitus on kuitenkin esitellä etenkin `map`- ja `fold`-funktioiden toimintaa, koska ne ovat yleisesti käytettyjä.

Ohjelma 3.11: Ohjelma havainnollistaa lambda-lausekkeiden ja kirjastoitujen toistorakenteiden käyttöä F#-kielellä.

```
let listOfInts =
   [
      for i in 1..100 do
         yield i
   ]

let result =
   listOfInts
   |> List.filter (fun i -> (i % 2) = 0)
   |> List.map (fun i -> 2 * i)
   |> List.fold (+) 0
```

Funktionaalisille kielille on ominaista funktioiden osittainen kutsuminen, tämä on mahdollista myös F#-kielessä. Osittaisen kutsumisen avulla voidaan olemassa olevista funktioista muodostaa uusia funktioita sitomalla vain osa argumenteista funktiokutsun yhteydessä. Näin tulokseksi saadaan uusi funktio, jolle kutsuttaessa tarvitsee antaa vain ne argumentit, joita ei vielä ole sidottu. Ohjelma 3.12 havainnollistaa osittaista kutsumista.

Ohjelma 3.12: Ohjelma havainnollistaa F#-kielen kurrisointia. Funktio multiplyBy2 kutsuu funktiota multiply osittain ja määrittää yhden parametrin.

```
let multiply x y = x * y

let multiplyBy2 = multiply 2

let result = multiplyBy2 5 // Result : 10
```

3.4 C#:n ja F#:n ominaisuuksien vertailu.

Kuten taulukosta 3.1 huomataan, C#:lla ja F#:lla on paljon yhteisiä piirteitä, jotka .NET-ympäristön hyödyntäminen tuo mukanaan. Tällaisia yhteisiä ominaisuuksia ovat esimerkiksi .NETin luokkakirjastot, tyyppijärjestelmä CTS ja roskienkeruu. F# on .NETin kirjastojen lisäksi oma F# Core -kirjasto. Molemmat kielet ovat myös yhteensopivia kaikkien muiden .NET-ympäristön kielien kanssa.

Kielet ovat kuitenkin pääparadigmojensa vuoksi hyvin erilaisia. Siinä missä C# on vahvasti olio-imperatiivinen, F# on funktionaalinen. Molemmat kielet mahdollistavat myös muiden paradigmojen mukaisen ohjelmoinnin. C#:ssa on mahdollista toteuttaa osia myös muilla paradigmoilla kuten esimerkiksi funktionaalisesti (Language Integrated Query, LINQ). Muita paradigmoja ei ole kuitenkaan tarkoitettu ohjelman rakenteen toteuttamiseen vaan niillä on tarkoitus helpottaa yksittäisten ominaisuuksien toteutusta.

F# puolestaan mahdollistaa funktionaalisen ohjelmoinnin lisäksi olio-imperatiivisen ohjelmoinnin. F# tukee olio-ohjelmointia ja periytymistä niin, että F#-ohjelma on mahdollista toteuttaa kokonaan olioilla. F#:n tyyppitarkistukset ovat kuitenkin tarkempia kuin C#:n. Tämän takia F#:ssa ei ole implisiittistä upcastia aliluokasta kantaluokkaan vaan tyyppimuunnos pitää tehdä eksplisiittisesti. Tämä tekee olioiden käsittelemisestä hankalampaa kuin C#:ssa.

Olioiden tyyppimuunnosten lisäksi F#:ssa kaikki muutkin tyyppimuunnokset ovat eksplisiittisiä. C# on F#:ia joustavampi tyyppimuunnoksissa, sillä muunnokset tehdään implisiittisesti silloin kun tietoa ei voi kadota tyyppimuunnoksessa. Implisiittinen tyyppimuunnos tehdään esimerkiksi silloin kun `int`-tyyppinen arvo sijoitetaan `long` tyyppiseen muuttujaan. Tietotyyppi `int` on nelitavuinen, etumerkillinen kokonaislukutyyppi, kun taas `long` on vastaava kahdeksan tavuinen tyyppi. Tällöin kaikki `int`-tyypin arvot voidaan esittää `long`-tyypillä, eikä tyyppimuunnoksessa voi siis kadota tietoa.

Molemmat kielet tukevat myös niin geneeristen funktioiden, arvojen ja jäsenmuuttujien kuin geneeristen tietorakenteidenkin määrittelyä. F#:n vaihtoehtotyypit voi myös määritellä geneerisiksi. Eksplisiittisesti geneerisen ohjelmakoodin lisäksi F# tekee funktioista geneerisiä automaattisesti silloin kun tarkkoja tyyppejä ei voida päätellä. Automaattisesta geneerisyydestä huolimatta ohjelmat ovat tyyppiturvallisia. Tämä on mahdollista koska tyyppipäättelyn ansiosta F#-ohjelmakoodissa ei tarvitse määritellä funktioiden ja arvojen tietotyyppejä. Tällöin kääntäjän on mahdollista yleistää funktioita.

Myös C#:ssa on tyyppipäättelijä. Se on ominaisuuksiltaan kuitenkin huomattavasti heikompi kuin F#:n typpipäättelijä. C#:n tyyppipäättelijälle kerrotaan eksplisiittisesti `var`-avainsanalla, että sen pitää päätellä muuttujan tyyppi. Tällöin muuttujan alkuarvoksi ei voi asettaa `null`-arvoa vaan se pitää alustaa halutun tyyppiseen arvoon. Tyyppipäättelijää ei voida C#:ssa myöskään hyödyntää funktioiden argumenttien tyyppien päättelyssä vaan ne pitää aina määritellä. Poikkeuksena tähän on lambda-lausekkeet, joiden parametrien tyyppejä ei tarvitse määritellä, vaan tyyppipäättelijä osaa päätellä ne.

F#:n tyyppipäättelijä on huomattavasti monipuolisempi. Sen pääteltäväksi voidaan jättää niin nimettyjen arvojen, muuttujien, funktioiden argumenttien ja pa-

luuarvojen tyypit. Ohjelmoijan tarvitsee vain harvoin teknisistä syistä eksplisiittisesti määritellä tyypit. Ohjelman luettavuuden kannalta se on kuitenkin suotavaa tilanteissa, joissa tyypit eivät ole kontekstista itsestään selviä.

Molemmissa kielissä ohjelmakoodi voidaan jakaa nimiavaruuksiin. Nimiavaruudet toimivat kummassakin kielessä samalla tavalla. Ne eivät voi suoraan sisältää funktioita tai arvoja vaan niiden pitää kuulua C#:ssa luokan määrittelyyn tai F#:ssa luokan, muun tietotyypin tai moduulin määrittelyyn. F#:ssa moduulit ovat ohjelmalohkoja, joilla voidaan ryhmitellä loogisesti yhteenkuuluva toiminnallisuus ja arvot yhdeksi kokonaisuudeksi. Toisin kuin olioista, moduuleista ei luoda instansseja. C#:ssa lähinnä moduulin käsitettä on staattiseksi määritellyt luokat.

Sekä C#:ssa että F#:ssa on mahdollista käyttää tapahtumia ja delegaatteja. Delegaateilla funktiokutsut voidaan esittää olioina. Tämä ei sinänsä tuo F#:iin mitään uutta, koska funktioita käsitellään arvoina muutenkin. Niitä voidaan kuitenkin käyttää .NETissä ja jotkin ohjelmointirajapinnat (engl. application programming interface, API) edellyttävät niiden käyttöä [31]. Tämän takia delegaatit on tuotu myös F#:iin.

Sen lisäksi että funktioita käsitellään F#:ssa datana kurrisoinnin ansiosta, niitä voidaan myös käsitellä C#:ia monipuolisemmin. C#:ssa funktiota kutsuttaessa on aina annettava kaikki argumentit. F#:ssa on mahdollista antaa vain osa argumenteista funktiota kutsuttaessa, jolloin tuloksena on uusi funktio, jolle argumenteiksi annetaan vain ne argumentit, joita ei jo aikaisemmin ole sidottu.

C#:ssa ei ole F#:lle ominaisia kirjastoituja kontrollirakenteita kuten `fold`, `map` ja `iter`. Osa F#:ssa yleisesti käytetyistä kirjastoiduista kontrolliraketneista on toteutettu osana LINQua. Esimerkiksi LINQun kuuluva `Select`-metodi vastaa F#:n `map`-funktiota.

3.5 Funktionaalisten kielten käyttö Atostekilla

Atostekissä funktionaalisia kieliä on käytetty useissa asiakasprojekteissa vuodesta 2008 alkaen. Funktionaalisia kieliä alettiin käyttämään koska haluttiin saada ohjelmien suunnittelusta tuottavampaa ja samalla parantaa ohjelmien laatua. Aluksi funktionaalisen kielen käyttöä kokeiltiin Haskellilla eräässä asiakkaalle tehdyssä pilottiprojektissa. Projektissa suunniteltiin myöhemmin toteutettavan ohjelmiston prototyyppi. Funktionaalisuudesta todettiin olevan niin paljon hyötyä, että lopullinen ohjelmakin päätettiin toteuttaa funktionaalisella kielellä. Haskellin sijaan lopullinen ohjelma toteutettiin kuitenkin F#:lla, koska vaatimuksena oli .NET-ympäristön käyttö.

Funktionaalisten kielten käyttö on todettu kannattavaksi erityisesti ohjelmissa, joissa hyödynnetään monimutkaisia tietorakenteita ja niitden käytöstä on tullut niin laajaa, että se on mahdollistanut tämän diplomityön tekemisen. Atostekilla ollaan

Taulukko 3.1: C#:n ja F#:n ominaisuuksien vertailu. Tähdellä (*) merkityt ominaisuudet ovat peräisin .NET-ympäristöstä.

	Kieli	
Ominaisuus	C#	F#
Paradigmat	**Olioparadigma**	**Funktionaalinen, olioparadigma**
Luokkakirjasto	.NET	.NET + F# Core
Ohjelmointikieliyhteensopivuus	.NET-ympäristön kielet	.NET-ympäristön kielet
Tyyppijärjestelmä*	CTS	CTS
Tyypitys (strukturaalinen / nominaalinen)	nominaalinen	nominaalinen
Tyyppipäättelijä	eksplisiittinen (`var`-avainsana)	implisiittinen
Tyyppimuunnokset: *oliot*	implisiittnen upcast, eksplisiittinen downcast	eksplisiittinen
Tyyppimuunnokset: *muut tietotyypit*	implisiittinen, jos tietoa ei katoa, muutoin eksplisiittinen	eksplisiittinen
Viitetyypit	kyllä	kyllä
Arvotyypit	kyllä	kyllä
Geneerisyys: *generics*	kyllä	kyllä
Geneerisyys: *funktioiden automaattinen yleistäminen*	ei	kyllä [29]
Osoittimet	kyllä	kyllä
Osoitinaritmetiikka	kyllä (`unsafe`-ohjelmalohkossa)	ei
LINQ	kyllä	kyllä (Query Expression [30])
Periytyminen	kyllä (ei moniperintää)	kyllä (ei moniperintää)
Rajapinnat	kyllä	kyllä
Roskienkeruu*	kyllä	kyllä
Reflection	kyllä	kyllä
Nimiavaruudet*	kyllä	kyllä
Moduulit	ei	kyllä
Delegaatit*	kyllä	kyllä
Tapahtumat*	kyllä	kyllä
Poikkeukset*	kyllä	kyllä
Kirjastoidut kontrollirakenteet	ei	kyllä
Korkean asteen funktiot	delegaatilla	kyllä
Kurrisointi	ei	kyllä

kiinnostuneita F#:sta, koska usein vaatimuksena on .NETin hyödyntäminen. F# ja .NET on myös havaittu hyvin toimivaksi ja tuottavaksi yhdistelmäksi. Funktionaalisten kielten käyttö ei ole Atostekilla rajoittunut kuitenkaan vain F#:iin. Muihin kuin .NET -ympäristöön tehtävissä ohjelmoinnissa on hyödynnetty Haskellia.

4. FUNKTIONAALISUUDEN JA F#-KIELEN KÄYTETTÄVYYDEN HAASTATTELUTUTKIMUS

Atostek Oy:ssä on käytetty useassa ohjelmistosuunniteluhankkeessa funktionaalisia ohjelmointikieliä. Tässä diplomityössä selvitetään Atostekilla työskentelevien ohjelmistosuunnitelijoiden kokemuksia funktionaalisten kielten käytettävyydestä. Työssä tarkastellaan, millaisia käytännön etuja ja haittoja funktionaalisten kielten käytöllä on verrattuna nykyään tavanomaisiin olio-imperatiivisiin kieliin. Erityisen kiinnostuksen kohteena on ollut F#:n käytettävyys verrattuna C#:iin.

Lisäksi tässä luvussa käsitellään haastatteluista saatuja tuloksia: esitellään numeerisia tuloksia, haastateltavien kommentteja ja analysoidaan saatuja tuloksia kokonaisuutena. Ensin käsitellään tuloksia C#:n ja F#:n käytettävyydestä neljästä näkökulmasta: opittavuus, tuottavuus, virheet ja tyytyväisyys. Lopuksi tarkastellaan haastattelujen tuloksia C#- ja F# -komponenttien yhdistämisestä käytännön ohjelmistoprojekteissa.

C#:n ja F#:n käytettävyyksien vertailun lisäksi työssä tarkastellaan F#:lla toteutettujen ohjelmistokomponenttien käytettävyyttä C#:lla tehdyistä komponenteista. Työssä on haluttu selvittää onko funktionaalisen F#-komponentin kääriminen C#:lla toteutettuun kääreeseen ollut kannattavaa ja mitä muita tapoja F#:lla toteutettujen komponenttien käyttämiseen C#-komponenteista on. Myös näitä asioita on tutkittu selvittämällä ohjelmistosuunnittelijoiden käytännön kokemuksia F#-komponenttien käyttämisestä C#-ohjelmakoodista.

4.1 Tutkimuskysymykset

Koska tavoitteena on kerätä ohjelmistosuunnittelijoiden kokemuksiin ja mielipiteisiin perustuvaa tietoa, haastattelu arvioitiin sopivaksi aineistonkeruumenetelmäksi. Haastattelumenetelmät voidaan jakaa strukturoituihin ja strukturoimattomiin sen perusteella, kuinka tarkkaan niiden rakenne on määritelty etukäteen. Strukturoimaton haastattelu antaa haastattelijalle mahdollisuuden reagoida ja kysyä lisäkysymyksiä tilanteen mukaisesti. Strukturoidun haastattelun kulku ja kysymykset on ennalta määrätty. Strukturoimaton haastattelu on strukturoitua haastattelua vaikeampi suorittaa, koska se vaatii enemmän osaamista, jotta haastattelu pysyy

aiheessa ja aikataulussa. Täysin strukturoitu haastattelu on haastattelijalle helppo suorittaa, koska hänen tarvitsee vain seurata ennalta laadittua suunnitelmaa. Hyödyllistä tietoa voi kuitenkin jäädä saamatta, koska haastattelija ei voi reagoida tilanteeseen.

Haastattelumenetelmäksi valittiin strukturoidun ja strukturoimattoman väliin sijoittuva puolistrukturoitu teemahaastattelu, koska se mahdollistaa haastateltavien vapaan kommentoinnin etukäteen valitun teeman puitteissa. Haastatteluja varten laadittiin kysymysrunko, kahden päätason teeman pohjalta. Ensimmäinen päätason teema jakautui vielä neljään alateemaan.

1. **Kielen käytettävyys.** Käytettävyysteeman tarkoitus oli selvittää ohjelmistosuunnittelijoiden kokemuksia kielen käytettävyydestä neljästä eri näkökulmasta. Näkökulmat on listattu alla. Kysymykset kysyttiin erikseen sekä C#:lle että F#:lle.

   (a) **Opittavuus.** Teeman tarkoituksena oli selvittää, kuinka vaikeaksi haastateltavat ovat kokeneet kielen opettelun verrattuna aikaisemmin opettelemiinsa kieliin. Tämän lisäksi tarkentavilla kysymyksillä haluttiin selvittää, mitkä seikat ovat vaikuttaneet oppimiseen.

   (b) **Tuottavuus.** Tuottavuus-teemalla haluttiin selvittää kuinka tehokkaaksi haastateltavat ovat kokeneet kielen ohjelmointityön tuottavuuden näkökulmasta. Lisäksi tarkentavilla kysymyksillä selvitettiin, mitkä kielen ominaisuudet haastateltavien mielestä vaikuttavat eniten tuottavuuteen.

   (c) **Virheet.** Virheet-teemalla selvitettiin, kuinka paljon ohjelmoijat kokevat tekevänsä virheitä, jotka johtuvat kielen ominaisuuksista. Tarkentavilla kysymyksillä kartoitettiin, mitkä ominaisuudet heidän mielestään aiheuttavat virheitä.

   (d) **Tyytyväisyys.** Tyytyväisyys-teemalla haluttiin selvittää, kuinka tyytyväisiä ohjelmoijat ovat kieleen yleisesti. Tarkentavilla kysymyksillä selvennettiin, mitkä kielten ominaisuudet aiheuttavat tyytyväisyyttä tai tyytymättömyyttä.

2. **F#:n ja C#:n käyttäminen samassa ohjelmassa.** Tällä teemalla haluttiin selvittää, millaiseksi ohjelmoijat kokevat F#:lla toteutettujen komponenttien käytön C# -ohjelmasta. Lisäksi haluttiin selvittää, onko F#:lla toteutettujen komponenttien kääriminen C# -luokan sisälle ollut kannattavaa ja onko se aiheuttanut virheitä ohjelmiin. Lisäksi pyrittiin kartoittamaan, mitä muita keinoja näillä kahdella kielellä toteutettujen komponenttien yhdistämisessä on.

Kysymysrunko muodostui teemoittain jaotellusta aihepiireistä ja niiden alle listatuista kysymyksistä. Kysymykset olivat numeerista arviota kysyviä tai mielipidekysymyksiä. Lisäksi molemmille edellä mainitulle tyypille oli tarkentavia kysymyksiä. Kysymysrunko on laadittu osana diplomityötä ja se on esitetty alla.

C# ja F# kielten käytettävyys

Opittavuus

- Miten vaikea kieli oli opetella aikaisemmin opettelemiisi verratuna? Arvioi asteikolla 1 - 5. *(1 helppo, 5 vaikea)*
- Miten opettelit kielen?
- Mikä oli vaikeaa kielen opettelussa? Mitä ongelmia kohtasit?
- Miten kielen opettelusta voisi tehdä helpompaa?
- Kestikö kauan ennen kuin pystyit käyttämään kieltä tehokkaasti?
- Mitkä ominaisuudet oli vaikea oppia?

Tuottavuus

- Onko kielen käyttäminen mielestäsi lisännyt tuottavuuttasi verrattuna aikaisempaan?
- Onko kielen opettelu muuttanut tapaasi ajatella ohjelmointia?
- Kuinka tehokkaaksi arvioit kielen kääntyvän ja suunnilleen toimivan ohjelman tuottamisessa? Arvioi asteikolla 1 - 5. *(1 tehoton, 5 tehokas)*
  - Mitkä kielen ominaisuudet vaikuttavat tuottavuuteen positiivisesti tai negatiivisesti?
- Ohjelmoinnissa tarvii usein lukea toisten kirjoittamaa koodia. Miten arvioisit kielellä kirjoitetun ohjelman luettavuutta? Arvioi asteikolla 1 - 5. *(1 huono, 5 hyvä)*
  - Mitkä ominaisuudet parantavat tai huonontavat luettavuutta?
- Kuinka hyvin kieli tukee modulaarisuutta? Arvioi asteikolla 1 - 5. *(1 huonosti, 5 hyvin)*
  - Kuinka hyvin ohjelman rakenne on hahmotettavissa valmiista ohjelmasta?
  - Mitkä ominaisuudet parantavat tai huonontavat rakenteen hahmotettavuutta?
- Kuinka vaikea kielellä kirjoitettujen ohjelmien rakennetta on muuttaa? Arvioi asteikolla 1 - 5. *(1 helppo - 5 vaikea)*
  - Mitä ongelmia olet kohdannut arkkitehtuuria muutettaessa?
  - Miten ongelmat olisi voitu välttää?

- Millaisiin tehtäviin suosittelisit kieltä? Miksi?
- Millaisiin tehtäviin et suosittelisi kielen käyttöä? Miksi et?

Virheet

- Kuinka paljon kielellä ohjelmoitaessa syntyy ennen kääntämistä tai kääntämisen aikana ilmeneviä virheitä? Arvioi asteikolla 1 - 5. *(1 vähän - 5 paljon)*
  - Mitkä asiat vaikuttavat virheiden määrään?
- Kuinka paljon kielellä virheitä jotka päätyvät lopputestaukseen tai aina asiakkaalle asti? Arvioi asteikolla 1 - 5. *(1vähän - 5 paljon)*
  - Mitkä asiat vaikuttavat virheiden määrään?
- Kuinka hyväksi arvioit kielen virheiden jäljitettävyyden näkökulmasta? Arvioi asteikolla 1 - 5. *(1 huono - 5 hyvä)*
  - Mitkä asiat vaikuttavat virheiden jäljitettävyyteen positiivisesti ja mitkä negatiivisesti?

Tyytyväisyys

- Kuinka tyytyväinen olet ollut kieleen? Arvioi asteikolla 1 - 5. *(1 tyytymätön, 5 tyytyväinen)*
- Mitkä asiat vaikuttavat eniten tyytyväisyyteen / tyytymättömyyteen? Nimeä 3 tärkeintä.
- Mitä kieltä käyttäisit mieluiten työssäsi?
- Mitkä ovat suurimmat haitat kielen käytössä?

C#- ja F#-komponenttien yhdistäminen

- Oletko käyttänyt C#-koodia F#-koodista?
  - Miten arvioisit sen käytettävyyttä?
  - Oletko käyttänyt F#-koodia C#-koodista?
  - Miten arvioisit sen käytettävyyttä?
- Atostekilla on useassa projektissa F#-toteutus kääritty C#-luokan sisälle, jotta todellinen toteutuskielen ominaisuudet peittyisivät käyttäjältä.
  - Miten arvioisit käärimistä? Onko siitä hyötyä?
  - Mitkä ovat sen suurimmat haitat / heikkoudet?
  - Kannattaako F#:ia käyttää vaikka joutuisi tekemään käärimisen veran lisätyötä?
  - Millaisiin tilanteisiin / ratkaisuihin kääriminen sopii?
  - Millaisiin tilanteisiin / ratkaisuihin kääriminen ei sovellu tai on turha?

  - Kuinka riskialttiiksi arvioit käärimisen? Arvioi asteikolla 1 - 5. *(1 ei-virhealtis, 5 virhealtis)*
  - Millaisia virheitä kääriminen on aiheuttanut?
- Miten F#- ja C#-koodia voisi muuten yhdistää kuin käärimällä? (tai käyttämällä suoraan?)
  - Vertaa ehdotusta käärimiseen. Miten työlääksi / hankalaksi / virhealttiiksi arvioit sen käärimiseen verrattuna?

4.2 Haastattelututkimuksen toteutus

Tutkimus toteutettiin haastattelemalla Atostekilla työskenteleviä ohjelmistosuunnittelijoita, joilla oli työtehtäviensä kautta kokemusta sekä C#:sta että F#:sta. Tutkimuksessa haastateltiin yhteensä yhdeksää Atostekin ohjelmistosuunnittelijaa. Tiedustelin kohderyhmän henkilöiltä suullisesta kiinnostuksesta osallistua haastattelututkimukseen. Kaikki yhdeksän ilmaisivat halukkuutensa osallistua haastatteluun.

Haastateltavista kahdeksan oli miehiä ja yksi nainen. Koulutukseltaan haastateltavat olivat hyvin samankaltaisia, seitsemällä oli diplomi-insinöörin tutkinto, yhdellä filosofian maisterin tutkinto ja yksi oli tekniikan ylioppilas. Kaikki diplomi-insinöörin tutkinnon suorittaneet olivat suorittaneet tutkintonsa Tampereen teknillisellä yliopistolla (TTY). Lisäksi tekniikan ylioppilas oli viimeistelemässä opintojaan samassa yliopistossa. Filosofian maisteri oli valmistunut Tampereen yliopistolta pääaineena tietojenkäsittely. TTY:llä opiskelleista kuudella oli ohjelmistotuotanto joko pää- tai sivuaineenaan. Kahden pääaine oli ohjelmistotiede. Haastateltavat olivat iältään 27 - 31 vuotiaita. Haastateltavien keski-ikä oli 28 ja mediaani 29 vuotta.

Haastattelut suoritettiin elo- ja syyskuussa 2013. Haastattelut tapahtuivat Atostekin toimistolla, ja ne nauhoitettiin älypuhelimen nauhoitustoiminnolla. Haastattelun alussa haastateltavilta kysyttiin lupa haastattelun nahoittamiseen ja kerrottiin, mihin haastattelut liittyvät ja miten haastattelu etenee teemoittain. Jälkikäteen arvioituna haastattelurunko toimi hyvin, mutta osa tarkentavista kysymyksistä olisi voinut olla paremmin muotoiltuja tulkinnanvaraisuuden vähentämiseksi. Useat haastateltavat mainitsivat miettineensä etukäteen asioita, joita nostaa esille haastattelussa ja nämä seikat nousivat esille haastatteluiden ensimmäisissä kysymyksissä ennen kuin ehdin esittämään asiaan liittyvää kysymystä. Kokonaisuudessa haastattelut toimivat kuitenkin hyvin eikä isoja ongelmia ilmennyt. Haastattelujen pituus vaihteli 50 minuutista kahteen tuntiin.

Kaikki haastateltavat ilmoittivat osanneensa jonkin muun olio-imperatiivisen ohjelmointikielen ennen C#:n opettelemista. Kaikki olivat ohjelmoineet aiemmin joko

C++:lla tai Javalla. Viisi haastatelluista kertoi, ettei heillä ollut kokemusta funktionaalisista ohjelmointikielistä ennen F#:iin tutustumista. Neljä haastatelluista mainitsi käyttäneensä jotain toista funktionaalista kieltä ennen F#:iin tutustumista. Heistäkin kolme korosti kokemuksen olleen erittäin vähäistä. Ennen F#:iin tutustumista nämä henkilöt kertoivat käyttäneensä Haskellia tai Scalaa. Yksi mainitsi ohjelmoineensa Prologilla.

Haastateltavat olivat tehneet huomattavasti enemmän työtunteja projekteissa, joissa oli käytetty C#:ia; keskimäärin 3730 tuntia. Projekteissa, joissa oli käytetty F#:ia, olivat haastateltavat keskimäärin tehneet 1950 tuntia. Haastateltavien työtuntimäärät sekä C#- että F#-projekteissa vaihtelivat 550 ja 5150 välillä. Työtuntimäärät on ilmoitettu kymmenen tarkkuuteen pyöristettyinä. Projekteissa käytetyt tuntimäärät ovat likimääräinen arvio haastateltavien kokemuksesta ohjelmointikielten käytössä. Se on kuitenkin paras objektiivinen mittari, joka oli käytettävissä. Lisäksi tunnit olivat helposti saatavilla Atostekin tuntienkirjausjärjestelmästä. Osassa projekteissa oli käytetty sekä F#:ia että C#:ia, jolloin näiden projektien tunnit laskettiin sekä C#- että F#-projektien tuntimääriin.

Haastateltavien anonymiteetin vuoksi haastateltavat on nimetty kirjaimella H ja numerolla. Numerointi on tehty satunnaisesti eikä perustu esimerkiksi haastattelujärjestykseen.

4.3 C#- ja F#-kielten käytettävyys

Tässä kohdassa käsitellään haastattelututkimuksen tulokset haastattelurungon ensimmäisen päätason teeman osalta. Teema oli jaettu neljään alakohtaan: opittavuus, tuottavuus, virheet ja tyytyväisyys. Kunkin alakohdan tulokset esitellään ja analysoidaan erikseen.

4.3.1 Opittavuus

Opittavuusteemalla selvitettiin, millaiseksi haastateltavat ovat kokeneet C#:n ja F#:n opettelun. Haastateltavia pyydettiin arvioimaan C#:n opettelun vaikeutta väliltä yhdestä viiteen. Vastaukset on nähtävissä taulukon 4.1 C#-sarakkeesta. Vastausten keskiarvo on 1,9 ja mediaani 2, C# arvioitiin siis erittäin helposti opeteltavaksi. F#:n vastaavat luvut ovat 3,7 ja 4, eli F# on huomattavasti vaikeampi opetella.

Lukuja vertaillessa tulee ottaa huomioon, että kaikilla haastatelluilla oli kokemusta jostain toisesta olio-imperatiivisesta ohjelmointikielestä ennen C#:n opettelua. Kaikki olivat ohjelmoineet aiemmin joko C++:lla tai Javalla. Tämän takia C#:n opettelu oli erittäin helppoa. F#:ia opetellessa vain neljällä haastateltavista oli kokemusta jostain toisesta funktionaalisesta ohjelmointikielestä. Heistäkin kolme mainitsi aikaisemman kokemuksen erittäin vähäiseksi. Aikaisempaa kokemuksia haasta-

teltavilla oli ollut Haskellista ja Scalasta. Koska lähes kenelläkään ei ollut kokemusta funktionaalisista kielistä tai kokemus oli erittäin vähäistä, joutuivat he kieltä opetellessaan omaksumaan myös erilaisen tavan ajatella ohjelmointia. Tämä selittää osaltaan, miksi F#:n opettelu koettiin vaikeaksi.

Taulukko 4.1: Vastaukset kysymykseen: "Miten vaikea kieli oli opetella aikaisemmin opettelemiisi verrattuna?"1 = helppo, 5 = vaikea.

	Kieli	
Haastateltava	C#	F#
H1	2	4
H2	2	4
H3	2	4
H4	1	3
H5	3	4
H6	2	3
H7	2	3
H8	2	4
H9	1	4
Mediaani	2	4
Keskiarvo	1,9	3,7

C#:n opettelun helppoudesta kertoo myös se, että kieli opeteltiin työskentelemällä sillä. Jotkut mainitsivat aloittaneensa olemassa olevan ohjelman muokkaamisella, toiset olivat puolestaan aloittaneet heti uuden ohjelmakoodin tuottamisella. Yksi haastateltavista mainitsi tukeutuneensa kirjaan. Ongelmatilanteissa he olivat tukeutuneet Internetistä löytyviin dokumentaatioihin ja kollegoihin.

C#:n ominaisuuksista vaikeita oppia ovat tapahtumat (engl. event), funktiodelegaatit, reflektio, LINQ ja arvo- ja viitevälitteisten parametrien ero. Lisäksi muutama haastateltava, jotka eivät olleet ennen ohjelmoineet kielellä, jossa on automaattinen muistinhallinta, kertoivat C#:n roskienkeruun olleen vaikea oppia. Nämä henkilöt olivat ohjelmoineet aikaisemmin C++:lla.

Haastateltavista kahdeksan oli opetellut F#:n perusteita jonkin kirjan avulla. Kielen kaikki olivat kuitenkin opetelleet lähinnä työskentelemällä sillä. Monet kertoivat ensin tehneensä muutoksia olemassa olevaan ohjelmaan ja vasta myöhemmin tehneensä uutta ohjelmakoodia. Osa haastateltavista oli tukeutunut myös Internetistä löytyviin tutoriaaleihin, kielen dokumentaatioon ja kollegojen apuun.

F#:n opettelussa verrattuna C#:n opetteluun suurin ero on se, että lähes kaikki olivat käyttäneet opettelun tukena kirjaa. Seitsemän mainitsikin F#:n opettelun haasteellisimmaksi osaksi uuteen paradigmaan liittyvän uuden ajattelutavan opettelun. Haastateltava H1 sanoi, että

> "Pitäisi olla kunnon kirja, joka selittäisi, mitä funktionaalisuus on ja missä olisi paljon esimerkkejä."

(H1)

F#:n opettelussa olennaista on funktionaalisen ajattelutavan omaksumisen lisäksi valmiiden kirjastofunktioiden ja kirjastoitujen kontrollirakenteiden opettelu ja omaksuminen. Esille nousivat erityisesti `fold`- ja `map`-funktiot yleisimmin käytettyinä ja siten tärkeinä oppia. Muita haasteita opettelussa olivat sijoituksen puuttuminen, funktioiden dataluonne, funktioiden kompositio ja monadit. Kaikki mainitut ominaisuudet ovat funktionaalisille kielille ominaisia, joten nämä kaikki haasteet liittyvät uuden ajattelutavan omaksumiseen.

F# ei ole ainoastaan funktionaalinen kieli vaikka funktionaalisuus onkin sen pääparadigma. Se mahdollistaa myös olio-imperatiivisen ohjelmoinnin. Haastateltava H4 sanoi imperatiivisten ominaisuuksien helpottaneen kielen opettelua. Haastateltava H9 puolestaan kertoi olio-imperatiivisten ominaisuuksien aiheuttaneen kieltä opetellessa väärinkäsityksiä ja funktionaalisuuden etujen menettämistä. Osa toivoi parempaa dokumentaatiota ja työkalutukea.

4.3.2 Tuottavuus

Tuottavuusteeman kysymyksillä selvitettiin, kuinka tehokasta C#:lla ja F#:lla työskentely on. Lisäksi selvittiin, mitkä kielten ominaisuudet vaikuttavat eniten työn tuottavuuteen. Taulukossa 4.2 on esitetty vastaukset kysymykseen, jolla selvitettiin, kuinka tehokkaita C# ja F# ovat tuotettaessa uutta ohjelmakoodia. Vastausten asteikko kulkee tehottomasta tehokkaaseen niin, että mitä suurempi luku sitä tehokkaammaksi kieli on arvioitu. Haastateltavat ovat kokeneet F#:n keskimäärin hieman tehokkaammaksi kuin C#. F#:lle vastausten keskiarvo on 3,7 ja C#:lle 3,6, eroa on siis vain yksi kymmenys. F#:n mediaani on kuitenkin 4 C#:n mediaanin ollessa 3.

Taulukko 4.2: Vastaukset kysymykseen: "Kuinka tehokkaaksi arvioit kielen kääntyvän ja suunnilleen toimivan ohjelman tuottamisessa? "1 = tehoton, 5 = tehokas.

	Kieli	
Haastateltava	C#	F#
H1	4	3
H2	3	4
H3	4	3
H4	3	5
H5	5	2
H6	3	4
H7	3	3
H8	4	4
H9	3	5
Mediaani	3	4
Keskiarvo	3,6	3,7

Lähes kaikki haastateltavat kokivat, että C#:lla ohjelmointi on tuottavampaa verrattuna heidän aikaisemmin osaamiinsa kieliin. Suurimmat syyt tuottavuuden parantumiseen ovat kuitenkin olleet hyvä kehitysympäristö eli IDE ja .NETin laajat kirjastot. Haastateltavat uskoivat työnsä tuottavuuden parantuneen myös F#:n käytön myötä, vaikka IDE-tuki kielelle onkin huonompi. Syitä tuottavuuden parantumiseen ovat olleet F#:n funktionaaliset ominaisuudet kuten automaattinen tyyppipäättely, sivuvaikutusten puuttuminen, `null`-arvojen puuttuminen, curry-muunnos ja `match`-valintarakenne. Lisäksi kirjastoidut kontrollirakenteet vähentävät virheitä, koska samoja toistorakenteita ei tarvitse kirjoittaa aina uudestaan. C#:n hyvä tuottavuus johtuu siis hyvistä työkaluista ja kirjastoista. F#:n tuottavuus puolestaan johtuu itse kielen ominaisuuksista ja sen tuottavuutta voitaisiin entisestään parantaa parantamalla työkalu- ja kirjastotukea.

Taulukossa 4.3 on tulokset kysymykseen, jolla selvitettiin kuinka helppolukuista C#:lla ja F#:lla kirjoitettu ohjelmakoodi on. Vastausten perusteella molemmilla kielillä kirjoitettu ohjelmakoodi on helppolukuista. Vastausten keskiarvo C#:lla oli 3,8 ja F#:lla 3,9, mediaanit olivat molemmilla kielillä 4. Tulosten perusteella ei siis saatu selvää eroa kielien luettavuudesta. Lähes kaikki haastateltavat mainitsivat, että molemmilla kielillä voi kirjoittaa epäselvää ohjelmakoodia, mutta hyviä muotoilukäytäntöjä noudattamalla molemmilla on mahdollista kirjoittaa helposti luettavia ohjelmia. Tämän lisäksi osa haastatelluista mainitsi F#:n olleen aluksi vaikealukuista, koska yksittäisten koodirivien sisältämä informaatiomäärä on suurempi kuin C#:ssa.

Vaikka tyyppipäättelyn todettiin parantavan F#:lla ohjelmoinnin tuottavuutta, sen todettiin myös hankaloittavan ohjelman lukemista. Mikäli ohjelmakoodista on jätetty muuttujien tyypit kirjoittamatta, pitää ohjelmaa lukea Visual Studiolla, jotta tyypit saadaan näkyviin. Tähän on kuitenkin ratkaisuna hyvien ohjelmointikäytäntöjen noudattaminen ja muuttujien tyyppien mainitsematta jättäminen vain silloin kun ne ovat asiayhteydestä itsestään selviä tai helposti pääteltäviä. C#:n yhteydessä pakollisia tyyppimäärittelyjä pidettiin lähinnä luettavuutta huonontavana tekijänä, koska runsaat tyyppimäärittelyt hukuttavat helposti varsinaisen asian.

Taulukoissa 4.4 ja 4.5 on esitetty tulokset siitä, kuinka hyvin C# ja F# tukevat ohjelman modulaarisuutta ja valmiin ohjelman rakenteen muuttamista. Modulaarisuuden osalta vastausten keskiarvo C#:lla on 4,0 ja F#:lla 3,4, mediaanit vastaavasti 4 ja 3. C# tukee siis paremmin ohjelman modulaarisuutta kuin F#. Kummassakin kielessä on olemassa lähes samat työkalut ohjelman modulaariseen suunnitteluun kuten nimiavaruudet, tiedostojako ja luokkarakenteet. Näiden lisäksi F#:ssa on moduulin käsite. Moduulit ja F#:n olio-ominaisuudet koettiin kuitenkin hankaliksi käyttää verrattuna C#:n olio-ominaisuuksiin.

Rakenteen muokattavuuden osalta vastausten keskiarvo C#:lle on 3,7 ja F#:lle 2,8

Taulukko 4.3: Vastaukset kysymykseen: "Miten arvioisit kielellä kirjoitetun ohjelman luettavuutta?"1 = huono, 5 hyvä.

	Kieli	
Haastateltava	C#	F#
H1	4	3
H2	4	5
H3	5	4
H4	5	5
H5	4	4
H6	3	4
H7	3	4
H8	3	2
H9	3	4
Mediaani	4	4
Keskiarvo	3,8	3,9

ja mediaanit 4 ja 3. Pienempi luku tarkoittaa rakenteen helpompaa muokattavuutta. F# -ohjelmien rakennetta on siis helpompi muuttaa kuin C# -ohjelmien. Tulos on F#:n kannalta hyvä, koska Visual Studion tuki ohjelmien muokkaamiselle on parempi C#:lle. Funktionaalisten periaatteiden mukaisesti kirjoitetun F# -ohjelman rakennetta on helpompi muokata, koska vältytään sivuvaikutusten aiheuttamilta virheiltä.

Taulukko 4.4: Vastaukset kysymykseen: "Kuinka hyvin kieli tukee modulaarisuutta?"1 = huonosti, 5 = hyvin.

	Kieli	
Haastateltava	C#	F#
H1	2	4
H2	3	3
H3	5	3
H4	5	5
H5	4	3
H6	5	3
H7	3	3
H8	4	3
H9	5	4
Mediaani	4	3
Keskiarvo	4,0	3,4

4.3.3 Virheet

Virheet-teemalla selvitettiin kuinka paljon kielillä syntyy ohjelmointivirheitä ja kuinka helppo virheet on korjata. Tuloksen odotettiin olevan, että F#:lla syntyy ohjelmoinnin aikana virheitä enemmän, mutta testaukseen ja asiakkaalle asti päätyvien

Taulukko 4.5: Vastaukset kysymykseen: "Kuinka vaikea kielellä kirjoitettujen ohjelmien rakennetta on muuttaa?"1 = helppo, 5 = vaikea. Yksi haastateltavista ei osannut vastata kysymykseen F#-kielen osalta.

	Kieli	
Haastateltava	C#	F#
H1	4	3
H2	3	4
H3	4	1
H4	4	4
H5	5	2
H6	4	2
H7	3	3
H8	3	-
H9	3	3
Mediaani	4	3
Keskiarvo	3,7	2,8

virheiden määrän olevan pienempi. Oletus perustuu siihen, että F#:n kääntäjä tekee enemmän tarkasteluja ja esimerkiksi tyyppijärjestelmä on tiukempi kuin C#:ssa. Tulos oli odotusten mukainen, mutta ohjelmoinninaikaisten virheiden ero on kielten välillä odotettua pienempi. Vastausten keskiarvot eroavat vain kolmella kymmenyksellä C#:n arvon ollessa 2,9 ja F#:n 3,2. Kummankin kielen mediaani on 3. Tulokset ovat nähtävissä taulukossa 4.6.

Testaukseen ja asiakkaalle päätyvien virheiden määrän kohdalla ero on selvempi. C#:n kohdalla keskiarvo on 3,0 ja mediaani 3. F#:n vastaavat arvot ovat 1,9 ja 2. Suurempi luku tarkoittaa enemmän virheitä. Vastaukset on nähtävissä taulukossa 4.7. Tulosten perusteella kielten välillä ei ole suurta eroa ohjelmoinnin aikana syntyvien virheiden määrässä. Valmiiseen ohjelmaan jää vähemmän virheitä, jos ohjelmointikielenä on ollut F#.

Kielten virheiden jäljitettävyyden osalta C# on selvästi parempi kuin F#. Tulosten keskiarvo C#:lle on 4,1 ja mediaani 4, vastaavat arvot F#:lle ovat 2,7 ja 3. Tulokset on esitetty taulukossa 4.8. Visual Studion työkalutuki virheiden jäljittämiselle on etenkin C#:lle erittäin hyvä kun taas F#:n osalta Visual Studion työkalutuki kaikilta osin C#:n tukea huonompi. Haastatellut kokivat virheiden jäljittämisen olevan helpompaa C#:lla kuin F#:lla. Tämä selittyy ainakin osin C#:n matalammalla abstraktiotasolla. Kun esimerkiksi kaikki toistorakenteet ja niiden vaiheet pitää kirjoittaa auki, voidaan suorituksen pysäytyskohtia asettaa tarkemmin juuri ohjelmoijan haluamiin kohtiin.

Vaikka F#:lla kirjastoidut toistorakenteet ja funktioiden kompositiot parantavat tuottavuutta ja vähentävät virheitä, vaikeuttavat ne kuitenkin virheiden jäljittämistä. Visual Studio ei mahdollista esimerkiksi pysäytyskohdan asettamista keskelle lambda-lauseketta, joilla usein annetaan kirjastoiduille toistorakenteille toistetta-

Taulukko 4.6: Vastaukset kysymykseen: "Kuinka paljon kielellä ohjelmoitaessa syntyy ennen kääntämistä tai kääntämisen aikana ilmeneviä virheitä?"1 = vähän, 5 = paljon.

	Kieli	
Haastateltava	C#	F#
H1	3	2
H2	3	3
H3	3	4
H4	2	4
H5	2	4
H6	3	2
H7	3	3
H8	2	4
H9	5	3
Mediaani	3	3
Keskiarvo	2,9	3,2

Taulukko 4.7: Vastaukset kysymykseen: "Kuinka paljon kielellä virheitä jotka päätyvät lopputestaukseen tai aina asiakkaalle asti?"1 = vähän, 5 = paljon.

	Kieli	
Haastateltava	C#	F#
H1	3	2
H2	3	2
H3	4	1
H4	4	2
H5	1	2
H6	3	2
H7	3	3
H8	3	2
H9	3	1
Mediaani	3	2
Keskiarvo	3,0	1,9

va toiminnallisuus. Funktioiden kompositiot hankaloittavat virheiden jäljittämistä, koska niitä käytettäessä Visual Studio ei mahdollista funktioiden paluuarvojen tarkastelua pysäytyskohtien avulla. Tämä rajoite voidaan kiertää sitomalla funktioiden paluuarvot väliaikaisiin nimiin, mutta se tekee virheiden etsimisestä työlästä etenkin pitkien kompositioiden osalta.

Myös `null`-arvojen puuttuminen ehkäisee virheitä funktionaalisten periaatteiden mukaan kirjoitetussa F#-ohjelmassa. C#:n osalta `null`-arvot koettiin yhtenä suurimmista virhelähteistä. Myös F#:n match-valintarakenne koettiin hyvänä ja virheitä ehkäisevänä, koska kääntäjä pakottaa käsittelemään kaikki `match`-valintarakenteen mahdolliset haarat.

Taulukko 4.8: Vastaukset kysymykseen: "Kuinka hyväksi arvioit kielen virheiden jäljitettävyyden näkökulmasta?"1 = huono, 5 = hyvä.

	Kieli	
Haastateltava	C#	F#
H1	4	2
H2	4	4
H3	5	3
H4	5	3
H5	4	2
H6	3	3
H7	4	2
H8	4	2
H9	4	3
Mediaani	4	3
Keskiarvo	4,1	2,7

4.3.4 Tyytyväisyys

Taulukossa 4.9 on esitetty haastateltujen yleinen tyytyväisyys C#:iin ja F#:iin. F#:in tulosten keskiarvo on 4,3 ja mediaani 5, C#:n kohdalla lukujen ollessa 3,4 ja 3. Haastateltavat ovat siis huomattavasti tyytyväisempiä F#:iin, vain kaksi haastateltavista oli tyytyväisempi C#:iin.

Taulukko 4.9: Vastaukset kysymykseen: "Kuinka tyytyväinen olet ollut kieleen?"1 = tyytymätön, 5 = tyytyväinen.

	Kieli	
Haastateltava	C#	F#
H1	3	4
H2	3	5
H3	4	5
H4	3	5
H5	4	3
H6	4	5
H7	3	4
H8	4	3
H9	3	5
Mediaani	3	5
Keskiarvo	3,4	4,3

C#:n kohdalla merkittävimpien tyytyväisyyteen vaikuttavat tekijät ovat hyvä kehitysympäristö ja työkalutuki, helppo opittavuus, virheiden jäljitettävyys, dokumentaatio ja valmiit kirjastot. F#:n kohdalla vastaavasti eniten tyytyväisyyttä tuottivat funktionaalisuus ja siihen liittyvät ominaisuudet kuten tyyppipäättely ja pattern matching. Lisäksi tyytyväisyyttä lisäsi vähäisempi virheiden määrä, tuottavuuden parantuminen, selkeä syntaksi ja se, että kaikki .NET-ympäristössä toimivat kir-

jastot ovat käytettävissä myös F#:sta. Tyytymättömyyttä synnytti C#:ia huonompi kehitysympäristö- ja työkalutuki.

Haastateltavilta tiedusteltiin, kumpi kielistä, C# vai F#, olisi heidän valintansa ohjelmointikieleksi työtehtävään, jos he saisivat täysin vapaasti valita. Kahdeksan yhdeksästä valitsisi mieluummin F#:n. Kuuden mielestä F#:ia kannattaa käyttää, vaikka joutuisi näkemään lisävaivaa C# -komponenttiin liittyäkseen. Yhden mielestä tämä ei kannata. Kaksi henkilöä ei osannut vastata kysymykseen. Vastaukset ovat nähtävissä taulukossa 4.10.

Taulukko 4.10: Haastateltavien valinnat C#:n ja F#:n väliltä (1) ja näkemys siitä, kannattaako F#:ia käyttää vaikka joutuisi näkemään lisävaivaa C# -komponenttiin liittyäkseen (2).

Haastateltava	1	2
H1	F#	Kyllä
H2	F#	Kyllä
H3	F#	Kyllä
H4	F#	Kyllä
H5	C#	Ei
H6	F#	Kyllä
H7	F#	-
H8	F#	-
H9	F#	Kyllä

4.4 C#- ja F#-komponenttien yhdistäminen

Haastatelluista kahdeksan oli käyttänyt C#:lla toteutettuja komponentteja F#:lla. .NET-ympäristön tarjoama mahdollisuus käyttää kaikkia .NET-virtuaalikoneelle käännettyjä komponentteja riippumatta toteutuskielestä koettiin hyödylliseksi. Täten F#:sakin voidaan hyödyntää kaikkia valmiita .NET-komponentteja. Tämä kuitenkin saattaa rikkoa F#:n funktionaalisuuden käyttämällä tilallista komponenttia, johon ei välttämättä huomata varautua F# -ohjelmassa. Samaan tapaan myös `null`-arvojen tarkastelu saattaa unohtua tällaisesta ohjelmasta ja näin johtaa virheisiin.

Kahdeksan oli käyttänyt F#:lla toteutettua komponenttia C#-ohjelmasta. Osa F#:lle ominaisista rakenteista on hankalasti käytettäviä ja omituisia C#:lla työskentelevän näkökulmasta. Esimerkiksi valintajoukko (engl. discriminated union) näkyy C#:ssa periytymishierarkiana ja F#:n moduulit näkyvät C#:ssa komponentissa staattisina luokkina. F#:lla toteutettujen komponenttien käyttäminen C#:sta onkin työlästä ja kahden kielen rajalla työskentely tulisi minimoida.

F#:lla ja C#:lla toteutettujen komponenttien yhdistämiseen onkin Atostekilla käytetty käärimistä. F#:lla toteutettu komponentti on käärityt C# -luokan sisälle. Näin

kahden kielen välillä työskentely on eristetty kääreeseen. Kuuden haastatellun mielestä F#:n käyttämisestä saadaan niin paljon etua, että sitä kannattaa käyttää vaikka joutuisi tekemään käärimisen verran lisätyötä. On kuitenkin tärkeää, ettei kääre aiheuta ohjelmaan lisää virheitä, koska se tuo komponenttiin uutta toiminnallisuutta. Taulukossa 4.11 on tulokset käärimisen virhealttiudesta. Niiden mukaan virhealttius on vähäinen, vastausten keskiarvo on 2,0 ja mediaani 2.

Taulukko 4.11: Vastaukset kysymykseen: "Kuinka riskialttiiksi arvioit käärimisen?"1 = ei lainkaan virhealtis, 5 = virhealtis. Haastateltavat, joilla ei ollut kokemusta F#:lla toteutetun komponentin käärimisestä C#-luokan sisään tai eivät muuten osanneet vastata kysymykseen on merkitty viivalla.

Haastateltava	Virhealttius
H1	1
H2	2
H3	2
H4	-
H5	-
H6	2
H7	-
H8	-
H9	3
Mediaani	2
Keskiarvo	2,0

Käärimisen lisäksi haastateltavat mainitsivat C#- ja F# -komponenttien yhdistämisen menetelmiksi

- F#-komponentin suoraan käyttämisen,
- konversion F#:sta tietorakenteista C#:ksi ja
- käärimisen toteuttamisen jo F#-komponentissa.

F#-komponentin suoraan käyttäminen todettiin työlääksi. Samoin konversion tekeminen F# -tietorakenteista C# -tietorakenteisiin todettiin työlääksi ja virhealttiiksi. Haastateltava H1 mainitsi, että tätä tekniikkaa on eräässä projektissa käytetty ja että siitä luovuttiin työläyden takia. Tämän lisäksi samaa toiminnallisuutta alkoi kertyä kumpaankin tietorakenteeseen, mikä puolestaan lisäsi virheitä ja työläyttä. Käärimisen toteuttaminen F#-komponentin puolella ei ole varsinaisesti kääresuunnittelumallin käyttöä. Siinä hyödynnetään F#:n olio-imperatiivisia ominaisuuksia ja suunnitellaan C#-komponentista käytettävät osat niin, että niiden käyttö on helppoa C#-komponentista. Tätä tekniikkaa ei ole kuitenkaan kokeiltu käytännössä.

5. ESIMERKKITOTEUTUS

Tässä luvussa esitellään funktionaalisen ja olio-imperatiivisen ohjelmoinnin eroja esimerkkiohjelman avulla. Esimerkkiohjelmassa on toteutettu punamusta puu sekä F#:lla että C#:lla. Kumpikin toteutus on toteutettu omissa komponenteissaan. Näiden lisäksi on toteutettu testiohjelma tietorakenteiden ja algoritmien testausta varten. Tässä luvussa on esitelty vain esimerkin kannalta olennaiset ohjelmakoodit. Esimerkkiohjelman lähdekoodi on kokonaisuudessaan liitteessä A

5.1 Punamusta puu

Punamustat puut (engl. red-black tree) ovat binäärihakupuita, joiden solmuille (engl. node) on asetettu väri musta tai punainen. Tämän lisäksi punamustan puun rakenteelle on asetettu rajoituksia eli invariantteja, jotka takaavat punamustan puun olevan tasapainotettu (engl. balanced). Tasapainoisuuden ansiosta perusoperaatiot kuten alkioiden lisäys, haku ja poisto ovat suoritettavissa $O(\lg n)$ -ajassa. Punamustat puut sopivat hyvin esimerkiksi joukko- (engl. set) ja sanakirja-tietorakenteiden (engl. map, dictionary) toteutukseen. [4, s. 273 - 301]

Cormen et al. [4, s. 273] on listannut viisi invarianttia, jotka täyttämällä punamusta puu pysyy tasapainossa. Kun puuhun lisätään tai siitä poistetaan alkioita invariantit saattavat mennä rikki. Tämän takia solmun lisäyksen tai poiston jälkeen puu tulee korjata niin, että se invariantit ovat jälleen voimassa.

1. Kaikki solmut ovat joko mustia tai punaisia.
2. Puun juurisolmu on musta.
3. Puun lehdet ovat mustia.
4. Punaisella solmulla voi olla vain mustia lapsisolmuja.
5. Polku jokaisesta solmusta sen lapsilehtiin sisältää yhtä monta mustaa solmua.

Ohjelmassa 5.1 on esitetty punamustan puun F#-toteutuksen runko. Siinä solmujen väri on toteutettu vaihtoehtotyyppinä `Color`. Puun toteuttava tietotyyppi on

puolestaan toteutettu geneerisenä vaihtoehtotyyppinä, jolle instanssin luomishetkellä annetaan tyyppi `'t`. Tyypiltä vaaditaan `comparison`-rajapinnan (eli vertailufunktion) toteuttamista. Puuta manipuloivat funktiot on toteutettu tyypin jäsenmetodeina. `RBTree`-moduulissa oleva funktiolla `empty`-luodaan tyhjä punamustapuu. Toteutus perustuu Chris Okasakin *Purely Functional Data Structures* -kirjassa [36, s. 24 - 29] esittelemään funktionaaliseen toteutukseen. F#-toteutus eroaa kuitenkin perinteisestä punamusta-puu toteutuksesta solmujen mahdollisten värien osalta. F#-toteutuksessa on käytetty punaisen ja mustan lisäksi tuplamusta -väriä. Tuplamustaa käytetään apuna solmun puusta poistavassa `delete` -funktiossa ja tuplamustia solmuja on olemassa solmun poiston ja puun uudelleen tasapainotuksen välisenä aikana. Tuplamusta -värin käyttö on mukailtu lähteestä [51].

Vastaava punamustan puun C#-toteutus perustuu Cormenin et al. *Introduction to Algorithms* -kirjassa [4, s. 273 - 301] esittelemiin proseduraalisiin algoritmilistauksiin. Oliototeutuksessa on kaksi geneeristä luokka. `Node` esittää puun solmuja ja `CsRBTree` itse puuta. Punamustaa puuta manipuloivat funktiot on toteutettu `CsRBTree`-luokan metodeina. Myös C#-toteutuksessa Punamustan puun tyypiltä `T` vaaditaan vertailurajapinnan toteuttamista. Luokkien rakenteet ilman metodien toteutuksia on esitetty ohjelmissa 5.2 ja 5.3.

Kuten kirjan esimerkissä, on C#-toteutuksessakin `null`-arvot korvattu *sentinel*-oliolla, joka tyyppiä `Node` ja jonka väri on musta. Muut jäsenten (`Parent`, `LeftChild`, `RightChild` ja `Element`) voivat olla mitä tahansa. Sitä käytetään esittämään puun lehtiä ja juuren esi-isää, joiden arvo muuten olisi `null`. Tällä tavalla vältytään `null`-vertailuilta ja ohjelmakoodista tulee helppolukuisempaa.

5.2 Alkion lisäys

C#-toteutuksessa alkion lisäykseen tarvitaan neljä funktiota: `Insert` (ohjelma 5.4), `InsertFixUp` (ohjelma 5.5) ja `LeftRotate` ja `RightRotate` (ohjelma 5.6). `Insert`-funktio on varsinainen lisäysfunktio. Sen tehtävänä on lisätä uusi alkio puussa oikeaan kohtaa. Alkion lisäys saattaa kuitenkin rikkoa puun invariantit. Koska uusi solmu väritetään punaiseksi, ainoat invariantit, jotka saattavat mennä rikki ovat kaksi ja neljä. Kutsumalla lisäysfunktion lopuksi `InsertFixup`-funktiota, korjataan invariantit. Korjausfunktio käyttää apunaan `LeftRotate`- ja `RightRotate`-funktioita puun manipuloinnissa. Korjaus saattaa kuitenkin rikkoa puun invariantin jostain toisesta kohtaa, `InsertFixup` korjaa puun invariantit lehdistä juureen päin kunnes puu taas täyttää sille asetetut ehdot.

Siinä missä C#-toteutus vaatii neljä funktiota ja 134 ohjelmakoodiriviä, F#-toteutukselle riittää kolme funktiota ja 21 ohjelmakoodiriviä (ohjelma 5.7). Varsinainen lisäysfunktio on triviaali, se kutsuu vain paikallista rekursiivista `ins`-funktiota, joka huolehtii alkion lisäyksestä oikeaan kohtaan, ja värittää tuloksena saadun puun

Ohjelma 5.1: Punamustan puun F#-toteutuksen runko.

```
module RBTree =

    type Color =
        | Red
        | Black
        | DoubleBlack

    type FsRBTree<'t when 't : comparison> =
        | Leaf of Color
        | Node of Color * FsRBTree<'t> * 't * FsRBTree<'t>
        with

            member this.isDoubleBlack =
                // ...

            member this.isBlack =
                // ...

            member this.blackify =
               // ...

            member this.contains x =
                // ...

            member this.insert x =
               // ...

            member this.depth =
                // ...

            member this.size =
                // ...

            member this.at n =
                // ...

            member this.min =
                // ...

            member this.delete x =
                // ..

            member this.blackHeight =
                // ...

            member this.Validate() =
                // ...

    let empty<'t when 't : comparison> : FsRBTree<'t> = Leaf Black
```

Ohjelma 5.2: Punamustan puun C#-toteutuksen solmua mallintavan luokan runko.

```
public class Node<T> where T : class, IComparable
{
    public enum NodeColor {
        Red, Black
    };

    public static readonly Node<T> Sentinel =
        new Node<T>(null) {
            Color = Node<T>.NodeColor.Black
        };

    public Node (T e) {
        // ...
    }

    public NodeColor Color { get; set; }

    public Node<T> Parent { get; set; }

    public Node<T> LeftChild { get; set; }

    public Node<T> RightChild { get; set; }

    public T Element { get; set; }

    public int GetDepth() {
        // ...
    }

    public int GetSize() {
        // ...
    }

    internal int ValidateNode() {
        // ...
    }

    internal bool Contains(T x) {
        // ...
    }

    internal Node<T> Find(object e) {
        // ...
    }
}
```

Ohjelma 5.3: Punamustan puun C#-toteutuksen runko. Rungosta on jätetty pois vain testaustarkoitukseen totetutettu `Validate`-funktio.

```
public class CsRBTree<T> where T : class, IComparable {
    private enum Color { Red, Black };

    public CsRBTree() {
        // ...
    }

    private Node<T> Root { get; set; }

    public int Depth {
        // ...
    }

    public int Size {
        // ...
    }

    public T ElementAt(int n) {
        // ...
    }

    public bool Contains(T x) {
        // ...
    }

    public void Insert(T e) {
        // ...
    }

    private void InsertFixUp(Node<T> z) {
        // ...
    }

    public T Delete(T e) {
        // ...
    }

    private Node<T> GetSuccessor(Node<T> x) {
        // ...
    }

    private Node<T> GetMinimum(Node<T> x) {
        // ...
    }

    private void DeleteFixup(Node<T> x) {
        // ...
    }

    private void LeftRotate(Node<T> x) {
        // ...
    }

    private void RightRotate(Node<T> x) {
        // ...
    }
}
```

Ohjelma 5.4: Funktio jolla C#-toteutuksessa puuhun lisätään alkio

```
public void Insert(T e)
{
    Node<T> z = new Node<T>(e);
    Node<T> y = Node<T>.Sentinel;
    Node<T> x = Root;
    while (x != Node<T>.Sentinel)
    {
        y = x;
        if (z.Element.CompareTo(x.Element) < 0)
        {
            x = x.LeftChild;
        }
        else
        {
            x = x.RightChild;
        }
    }
    z.Parent = y;
    if (y == Node<T>.Sentinel)
    {
        Root = z;
    }
    else if (z.Element.CompareTo(y.Element) < 0)
    {
        y.LeftChild = z;
    }
    else
    {
        y.RightChild = z;
    }
    z.LeftChild = Node<T>.Sentinel;
    z.RightChild = Node<T>.Sentinel;
    z.Color = Node<T>.NodeColor.Red;
    InsertFixUp(z);
}
```

Ohjelma 5.5: Funktio, jolla C#-toteutuksessa korjataan puun rakenne alkion lisäyksen jälkeen. Totetuksesta on näkyvissä vain osa, koska ulomman `else`-lohkon toteutus on olennaisilta osin saman kaltainen kuin `if`-haaran; vasen ja oikea vain vaihtavat paikkoja.

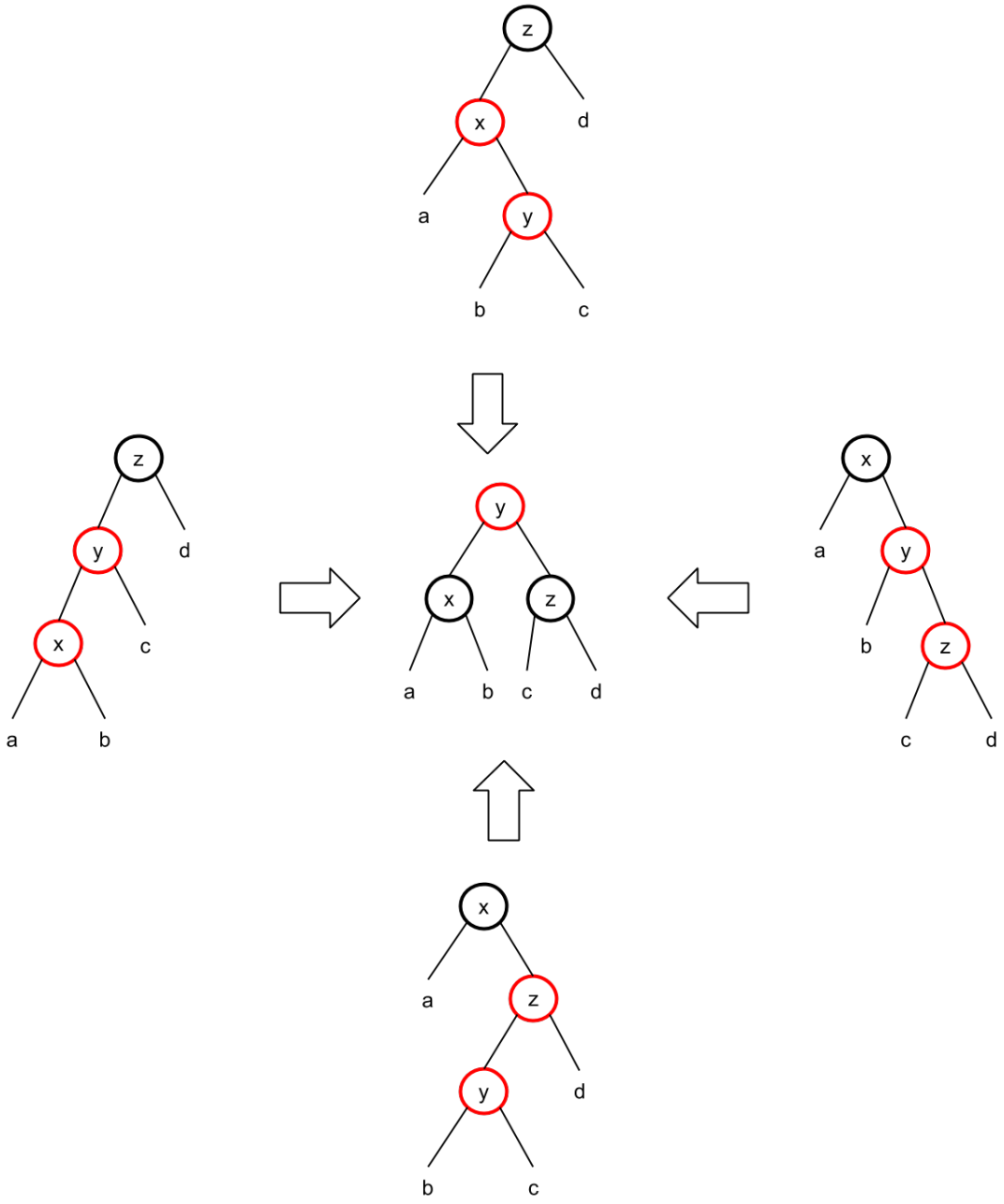
```
private void InsertFixUp(Node<T> z)
{
    while(z.Parent.Color == Node<T>.NodeColor.Red)
    {
        if(z.Parent == z.Parent.Parent.LeftChild)
        {
            Node<T> y = z.Parent.Parent.RightChild;
            if(y.Color == Node<T>.NodeColor.Red)
            {
                z.Parent.Color = Node<T>.NodeColor.Black;
                y.Color = Node<T>.NodeColor.Black;
                z.Parent.Parent.Color = Node<T>.NodeColor.Red;
                z = z.Parent.Parent;
            }
            else
            {
                if(z == z.Parent.RightChild)
                {
                    z = z.Parent;
                    LeftRotate(z);
                }
                z.Parent.Color = Node<T>.NodeColor.Black;
                z.Parent.Parent.Color = Node<T>.NodeColor.Red;
                RightRotate(z.Parent.Parent);
            }
        }
        else
        {
            // ...
        }
    }
    Root.Color = Node<T>.NodeColor.Black;
}
```

Ohjelma 5.6: `LeftRotate` ja `RightRotate` ovat funktioita, joilla puun rakennetta muokaan paikallisesti. Kun `LeftRotate` tehdään solmulle `x`, se siirtää `x`:n oikean laspisolmun `x`:n tilalle, tekee `x`:stä `y`:n vasemman lapsisolmun ja `y`:n vasemmasta lapsisolmusta `x`:n oikean lapsisolmun. `RightRotate` tekee saman kuin `LeftRotate`, mutta symmetrisesti toisin päin.

```
private void LeftRotate(Node<T> x)
{
    Node<T> y = x.RightChild;
    x.RightChild = y.LeftChild;
    if (y.LeftChild != Node<T>.Sentinel)
    {
        y.LeftChild.Parent = x;
    }
    y.Parent = x.Parent;
    if (x.Parent == Node<T>.Sentinel)
    {
        Root = y;
    }
    else if (x == x.Parent.LeftChild)
    {
        x.Parent.LeftChild = y;
    }
    else
    {
        x.Parent.RightChild = y;
    }
    y.LeftChild = x;
    x.Parent = y;
}

private void RightRotate(Node<T> x)
{
    // Sama kuin LeftRotate, mutta oikea
    // ja vasen toisin päin.
}
```

juuren mustaksi. Lisäyksen rikkomat invariantit korjataan paikallisella `balance`-funktiolla, joka vastaa C#-toteutuksen `InsertFixup`-, `LeftRotate`- ja `RightRotate`-funktioita. Erona C#-toteutukseen `insert`-funktio palauttaa paluuarvonaan kokonaan uuden puun, johon uusi alkio on lisätty. Vanha puu jää muuttumattomaksi. C#-totetutus olio-imperatiivisen paradigman mukaisesti muokkaa puuolion tilaa, jolloin vanha rakenne tuhoutuu.



Kuva 5.1: Kuva havainnollistaa, miten ohjelman 5.7 `balance`-funktio poistaa ylimääräisen punaisen solmun alkion lisäyksen jälkeen. Mukailtu lähteestä [36, s. 27].

Ohjelma 5.7: Funktio lisää alkion punamustan puun F#-toteutukseen.

```
member this.insert x =
    let balance tree =
        match tree with
        | Node(Black,Node(Red,Node(Red,a,x,b),y,c),z,d)
        | Node(Black,Node(Red,a,x,Node(Red,b,y,c)),z,d)
        | Node(Black,a,x,Node(Red,Node(Red,b,y,c),z,d))
        | Node(Black,a,x,Node(Red,b,y,Node(Red,c,z,d))) ->
            Node(Red,Node(Black,a,x,b),y,Node(Black,c,z,d))
        | _ -> tree

    let rec ins tree =
        match tree with
        | Leaf _ ->
            Node(Red,Leaf Black,x,Leaf Black)
        | Node(color,l,y,r) when x < y ->
            balance (Node(color,(ins l),y,r))
        | Node(color,l,y,r) when x > y ->
            balance (Node(color,l,y,(ins r)))
        | _ -> tree

    match ins this with
    | Node(_,l,e,r) -> Node(Black,l,e,r)
    | this -> this
```

5.3 Alkion poisto

Alkion poisto on sekä C#- että F#-toteutuksessa hieman monimutkaisempi kuin alkion lisäys. C#-toteutuksessa alkion poistoon vaaditaan viisi funktiota: `Delete` (ohjelma 5.8), `DeleteFixup` (ohjelma 5.10), `GetSuccessor` (ohjelma 5.9) ja alkion lisäyksessäkin käytetyt `LeftRotate` ja `RightRotate`. `GetSuccessor`-funktio on yksinkertainen apufunktio, jonka avulla etsitään solmua järjestyksessä seuraava suurempi solmu.

`Delete`-funktio on periaatteeltaan hyvin yksinkertainen, se etsii poistettavan solmun, poistaa sen ja sen jälkeen korjaa puun invariantit `DeleteFixup`-funktiolla. `Insert`-funktion tapaan `Delete`-funktio muokkaa puuolion rakennetta ja poistoa edeltänyt puun tila tuhoutuu. Paluuarvonaan se palauttaa poistetun solmun. `DeleteFixup`-funktion toimintaperiaate on sama kuin `InsertFixup`:n, se tarkistaa onko puun invarianti rikki poistokohdasta ja korjaa sen sitten lehdistä juurta kohti. Solmujen järjestystä se manipuloi `LeftRotate`- ja `RightRotate`-funktioiden avulla.

Toisin kuin muu osa F#-toteutuksesta, joka perustuu Okasakin esimerkkeihin lähteessä [36], `delete`-funktion (ohjelmat 5.11 ja 5.12) toteutus on mukailtu lähteestä [51]. Sen idea alkion poiston osalta on yksinertainen: etsi poistettava solmu ja poista se:

- Mikäli poistettava solmu on lehti, se voidaan vain poistaa.

Ohjelma 5.8: `Delete`-funktio poistaa halutun alkion puusta ja kutsuu sen jälkeen `DeleteFixup`-funktiota korjatakseen puun invariantit.

```
public T Delete(T e)
{
    Node<T> z = Root.Find(e);
    if (z == null)
    {
        throw new ArgumentException("Value␣not␣found.");
    }

    Node<T> y = Node<T>.Sentinel;
    if (z.LeftChild == Node<T>.Sentinel ||
        z.RightChild == Node<T>.Sentinel)
    {
        y = z;
    }
    else
    {
        y = GetSuccessor(z);
    }

    Node<T> x = Node<T>.Sentinel;
    if (y.LeftChild != Node<T>.Sentinel)
    {
        x = y.LeftChild;
    }
    else
    {
        x = y.RightChild;
    }

    x.Parent = y.Parent;

    if (y.Parent == Node<T>.Sentinel)
    {
        Root = x;
    }
    else if (y == y.Parent.LeftChild)
    {
        y.Parent.LeftChild = x;
    }
    else
    {
        y.Parent.RightChild = x;
    }

    if (y != x)
    {
        z.Element = y.Element;
    }

    if (y.Color == Node<T>.NodeColor.Black)
    {
        DeleteFixup(x);
    }
    return y.Element;
}
```

Ohjelma 5.9: `GetSuccessor`-funktio etsii solmua yhtä suuremman solmun.

```
private Node<T> GetSuccessor(Node<T> x)
{
    if (x.RightChild != Node<T>.Sentinel)
    {
        return GetMinimum(x.RightChild);
    }

    Node<T> y = x.Parent;
    while (y != Node<T>.Sentinel && x == y.RightChild)
    {
        x = y;
        y = y.Parent;
    }
    return y;
}
```

- Mikäli sillä on vain yksi lapsisolmu, poistetaan solmu ja asetetaan lapsisolmu poistetun tilalle.
- Mikäli poistettavalla solmulla on molemmat lapsipuut, etsitään oikean lapsipuun pienin solmu ja kopioidaan tämä poistetun solmun tilalle. Tämän jälkeen pitää vain poistaa kopioitu solmu sen alkuperäisestä sijainnista.

Jos poistettu solmu oli musta, "tummennetaan" sen tilalle asetetun solmun väriä, jolloin sen väristä tulee musta, jos se oli punainen ja tupamusta, jos se oli musta. Tämän jälkeen `balance`-funtion tehtäväksi jää poistaa tuplamusta väri ja suorittaa tarvittava tasapainotus.

5.4 C#- ja F#-toteutusten vertailu

Helpoimmin huomattava ero C#- ja F#-toteutusten välillä on ohjelmakoodirivien määrässä. C#-toteutuksessa puuluokan ja solmuluokan ohjelmakooditiedostojen (RBTree.cs ja Node.cs) yhteenlaskettu rivimäärä on 470. F#-toteutus on sijoitettu yhteen luokkaan, jonka rivimäärä on 118. C#-toteutus vaatii siis noin neljä kertaa niin paljon ohjelmakoodirivejä kuin F#-toteutus. Rivimäärät pitävät sisällään kumpaankin toteutukseen toteutuksen testaamista helpottavat validointifunktiot, jotka tarkistavat puun invariantit. C#-toteutuksen osalta testikoodia on 33 riviä eli noin 7% koko ohjelmasta. F#-toteutuksen osalta testikoodia on 18 riviä eli noin 15%.

C#-toteutuksessa punamustan puun solmujen tyyppinä ei voida käyttää perustietotyyppiä kuten esimerkiksi `int`, koska tyypin tulee olla viitetyyppiä. Vaatimus viitetyypistä on peräisin siitä, että tyhjään arvoon tulee voida viitata `null`-viittauksella. Mikäli perustietotyyppejä halutaan käyttää, pitää ne kääriä luokan sisään. Esimerkkiohjelmassa käytetty kääre on hyvin yksinkertainen ja toteuttaa vain `Node`- luokan

Ohjelma 5.10: `DeleteFixup` korjaa puun invariantit alkion poiston jälkeen.

```
private void DeleteFixup(Node<T> x)
{
    while (x != Root && x.Color == Node<T>.NodeColor.Black)
    {
        if (x == x.Parent.LeftChild)
        {
            Node<T> w = x.Parent.RightChild;
            if (w.Color == Node<T>.NodeColor.Red)
            {
                w.Color = Node<T>.NodeColor.Black;
                x.Parent.Color = Node<T>.NodeColor.Red;
                LeftRotate(x.Parent);
                w = x.Parent.RightChild;
            }
            if (w.LeftChild.Color == Node<T>.NodeColor.Black &&
                w.RightChild.Color == Node<T>.NodeColor.Black)
            {
                w.Color = Node<T>.NodeColor.Red;
                x = x.Parent;
            }
            else
            {
                if (w.RightChild.Color == Node<T>.NodeColor.Black)
                {
                    w.LeftChild.Color = Node<T>.NodeColor.Black;
                    w.Color = Node<T>.NodeColor.Black;
                    RightRotate(w);
                    w = x.Parent.RightChild;
                }
                w.Color = x.Parent.Color;
                x.Parent.Color = Node<T>.NodeColor.Black;
                w.RightChild.Color = Node<T>.NodeColor.Black;
                LeftRotate(x.Parent);
                x = Root;
            }
        }
        else
        {
            // Sama kuin if-haara, mutta oikea ja
            // vasen toisin päin.
        }
    }
    x.Color = Node<T>.NodeColor.Black;
}
```

Ohjelma 5.11: `Delete`-funktio poistaa alkion puusta ja korjaa sen jälkeen puun invariantit kutsumalla `balance`-funktiota. Selkeyden vuoksi `balance`-funktion toteutus on estetty ohjelmassa 5.12.

```
member this.delete x =
    let rec balance tree =
        // ...

    let rec del tree =
        match tree with
        | Node(c,l,e,r) when x < e ->
            balance (Node(c,(del l),e,r))
        | Node(c,l,e,r) when x > e ->
            balance (Node(c,l,e,(del r)))
        | Node(Black,Leaf Black,e,Leaf Black) when x = e ->
            Leaf DoubleBlack
        | Node(Red,Leaf Black,e,Leaf Black) when x = e ->
            Leaf Black
        | Node(DoubleBlack,Leaf Black,e,Leaf Black) when x = e ->
            Leaf Red
        | Node(c,Node(cc,cl,ce,cr),e,Leaf _)
        | Node(c,Leaf _,e,Node(cc,cl,ce,cr)) when x = e ->
            match c,cc with
            | Red, Black
            | Black, Red -> Node(Black,cl,ce,cr)
            | Black, Black -> Node(DoubleBlack,cl,ce,cr)
            | _,_ -> Node(Red,cl,ce,cr)
        | Node(c,l,e,r) when x = e ->
            let newValue = r.min
            balance (Node(c,l,newValue,(r.delete newValue)))
        | t -> t

    match del this with
    | Node(_,l,e,r) -> Node(Black,l,e,r)
    | t -> t
```

Ohjelma 5.12: Rekursiivinen `balance`-funktio poistaa tuplamusta värin puusta ja korjaa puun invariantit poiston jälkeen.

```
member this.delete x =
    let rec balance tree =
        match tree with
        | Node(Black,pl,pe,Node(Red,rl,re,rr))
            when pl.isDoubleBlack ->
            balance (Node(Black,balance (Node(Red,pl,pe,rl)),re,rr))
        | Node(Red,pl,pe,Node(Black,rl,re,rr))
            when pl.isDoubleBlack && rl.isBlack && rr.isBlack ->
            Node(Black,pl.blackify,pe,Node(Red,rl,re,rr))
        | Node(Black,pl,pe,Node(Black,rl,re,rr))
            when pl.isDoubleBlack && rl.isBlack && rr.isBlack ->
            Node(DoubleBlack,pl.blackify,pe,Node(Red,rl,re,rr))
        | Node(DoubleBlack,pl,pe,Node(Black,rl,re,rr))
            when pl.isDoubleBlack && rl.isBlack && rr.isBlack ->
            Node(Red,pl.blackify,pe,Node(Red,rl,re,rr))
        | Node(pc,pl,pe,Node(Black,Node(_,rll,rle,rlr),re,rr))
            when pl.isDoubleBlack && rr.isBlack ->
            balance (Node(pc,pl,pe,
                          Node(Black,rll,rle,Node(Red,rll,re,rr))))
        | Node(pc,pl,pe,Node(Black,rl,re,rr))
            when pl.isDoubleBlack ->
            Node(pc,Node(Black,pl.blackify,pe,rl),re,rr.blackify)
        | Node(Black,Node(Red,ll,le,lr),pe,pr)
            when pr.isDoubleBlack ->
            balance (Node(Black,ll,le,balance (Node(Red,lr,pe,pr))))
        | Node(Red,Node(Black,ll,le,lr),pe,pr)
            when pr.isDoubleBlack && ll.isBlack && lr.isBlack ->
            Node(Black,Node(Red,ll,le,lr),pe,pr.blackify)
        | Node(Black,Node(Black,ll,le,lr),pe,pr)
            when pr.isDoubleBlack && ll.isBlack && lr.isBlack ->
            Node(DoubleBlack,Node(Red,ll,le,lr),pe,pr.blackify)
        | Node(DoubleBlack,Node(Black,ll,le,lr),pe,pr)
            when pr.isDoubleBlack && ll.isBlack && lr.isBlack ->
            Node(Red,Node(Red,ll,le,lr),pe,pr.blackify)
        | Node(pc,Node(Black,ll,le,Node(_,lrl,lre,lrr)),pe,pr)
            when pr.isDoubleBlack && ll.isBlack ->
            balance (Node(pc,
                          Node(Black,Node(Red,ll,le,lrl),lre,lrr),pe,pr))
        | Node(pc,Node(Black,ll,le,lr),pe,pr) when pr.isDoubleBlack ->
            Node(pc,ll.blackify,le,Node(Black,lr,pe,pr.blackify))
        | t -> t
```

vaatiman `IComparable`-rajapinnan. Sen toteutus on sijoitettu omaan tiedostoonsa (TestObject.cs), jonka pituus on 38 riviä. F#-toteutuksessa tällaista rajoitusta ei ole.

Merkittävin ero toteutusten välillä on kuitenkin ohjelman luettavuudessa ja ymmärrettävyydessä. Ero ohjelman ymmärrettävyydessä on helpoiten huomattavissa uuden solmun lisäävien funktioiden kohdalla. Siinä missä C#-toteutus vaatii neljä funktiota, F#-toteutus vaatii vain kolme. Ymmärrettävyyden ero korostuu erityisesti solmun lisäyksen jälkeen invariantit voimaan saattavien funktioiden kohdalla. F#-ohjelman `balance`-funktiossa on kuvattu kaikki neljä tapaa, joilla puun invariantit voivat olla rikki alkion lisäyksen jälkeen. Näiden alla on rivi, joka kertoo, miten solmut asetellaan, jotta puun invariantit olisivat jälleen voimassa. Kun funktion toteutusta vertaa kuvaan 5.1, on siitä helppo nähdä, että toteutus on oikein.

C#-toteutuksen osalta alkion lisäävien ja puun invariantit voimaan saattavien funktioiden voimaan oikeellisuudesta ei ole yhtä helppo varmistua. Ohjelmaa lukevan henkilön tulee suorittaa ohjelmaa mielessään ja muistaa, miten ohjelma on edennyt ehtolauseiden ja toistorakenteiden haarauttamien mahdollisten suorituspolkujen läpi. Samasta syystä ohjelman toimintaa on vaikea visualisoida ja siten helpottaa sen ymmärtämistä.

Punamustan puun määrittelystä johtuen molemmat toteutukset ovat solmun poiston osalta monimutkaisempia kuin solmun lisäyksessä. Erityisesti F#-toteutus näyttää huomattavan monimutkaiselta verrattuna solmun lisäykseen. Poistofunktio `del` on rekursiivinen ja siinä on esitetty kaikki mahdolliset tilanteet ja määritelty, miten poisto kussakin tilanteessa suoritetaan. Tasapainotusfunktio `balance` on myös huomattavan monimutkainen verrattuna tasapainotukseen solmun lisäyksen tapauksessa. Sekin on kuitenkin rakennettu samalla periaatteella. Valintarakenteessa on listattu kaikki mahdollisuudet, joilla puu voi olla epätasapainossa poiston jälkeen, ja määritelty, miten puu kussakin tapauksessa saadaan takaisin tasapainoon.

Lisäksi F#:n kääntäjä pitää huolen siitä, että `match`-rakenteessa käsitellään kaikki mahdolliset tapaukset. Jos kaikkia mahdollisia vaihtoehtoja ei ole määritelty, kääntäjä antaa siitä virheilmoituksen eikä ohjelma käänny. C#-toteutuksessa ei voida samalla tavalla mekaanisesti varmistua siitä, että kaikki `if`-, `if else`- ja `else`-haarat on määritelty. Koska solmun poiston on monimutkaisempi myös C#:n osalta, on myös ohjelman ymmärtäminen ja sen mielessä suorittaminenkin entisät vaikeampaa.

F#-toteutuksessa ei ohjelmoijan tarvitse huolehtia `NullReferenceException`-poikkeuksista, koska tyhjät arvot on ilmaistu `Option`-rakenteen `None`-arvolla. C#-toteutuksessa suurin osa `null`-arvon tarkastuksista on onnistuttu välttämään `Sentinel`-arvon käytöllä, mutta siitäkin huolimatta esimerkiksi alkion poiston yhteydessä pitää varautua poikkeukseen tilanteessa, jossa yritetään poistaa arvoa, jota puusta ei löydy.

6. YHTEENVETO

Ohjelmointikielten toimintamallit voidaan jakaa imperatiiviseen ja deklaratiiviseen paradigmaan. Paradigmat eroavat toisistaan sen suhteen, miten ne lähestyvät ohjelmointia. Imperatiivisen paradigman kielillä kuvataan, mitä operaatioita tietokoneen tulisi suorittaa ja missä järjestyksessä. Deklaratiivisen paradigman kielillä määritellään laskennan osat ja niiden väliset riippuvuudet ja suoritusjärjestys jätetään tietokoneen pääteltäväksi.

Oliokielet ovat imperatiivisia ohjelmointikieliä. Niissä ohjelmat kootaan reaalimaailman esineitä ja asioita mallintavista olioista. Toteutuksen yksityiskohdat kapseloidaan olion sisään ja oliota käytetään ja sen tilaa muokataan olion tarjoaman rajapinnan kautta. Funktionaaliset kielet ovat deklaratiivisia ohjelmointikieliä, jotka perustuvat lambda-kalkyylille. Lambda-kalkyyli kuvaa laskentaa funktioiden avulla. Olio-imperatiivisin kieliin nähden ne tarjoavat funktioiden käsittelyyn monia uusia työkaluja kuten funktioiden käsittely datana, lambda-lausekkeet ja funktioiden kompositio. Funktionaalisille kielille on myös ominaista kirjastoidut kontrollirakenteet. Funktionaaliset ohjelmat ovat tilattomia. Tämä helpottaa ohjelmien kirjoittamista ja lukemista, koska kirjoittajan ja lukijan ei tarvitse huolehtia suoritusjärjestyksestä vaan se on kääntäjän vastuulla.

Tässä työssä käsiteltiin erityisesti funktionaalista F#:ia ja olio-imperatiivista C#:ia. Molemmat ovat Microsoftin .NET-ympäristössä käytettäviä kieliä. .NET-ympäristö helpottaa eri kielillä toteutettujen ohjelmakomponenttien yhdistämistä, koska kaikki sen CIL-välikielelle käännettävät kielet ovat virtuaalikoneen tasolla yhteensopivia. Se tarjoaa myös yhteisen tyyppijärjestelmän CTS, jonka ansiosta kaikissa .NET-ympäristön kielissä on samat perustietotyypit. .NET ei kuitenkaan peitä kielten eri paradigmoista johtuvia erilaisuuksia.

Haastattelututkimuksessa arvioitiin C#:n ja F#:n käytettävyyttä neljästä näkökulmasta: opittavuus, tuottavuus, virheet ja tyytyväisyys. Saatujen tulosten perusteella C# on F#:ia helpompi oppia. C#:n opettelua helpottaa ohjelmistosuunnittelijoiden aikaisempi kokemus oliokielistä. F#:n opettelusta tekee haasteellisempaa funktionaalisuus. Vain harvalla haastatellulla oli kokemusta funktionaalisista kielistä tai kokemus oli vähäistä. Samasta syystä F#:ia opetellessaan moni ohjelmistosuunnittelija joutuu samalla omaksumaan uuden paradigman ja sen mukaisen tavan ohjelmoida.

Ohjelmistosuunnittelijat kokivat, että C#:n käyttäminen paransi tuottavuutta heidän aikaisemmin kokeilemiinsa kieliin nähden. F# arvioitiin hieman C#:ia tuottavammaksi. Siinä missä C#:n tuottavuutta eniten lisäsi Visual Studio -kehitysympäristö, syy F#:n hyvään tuottavuuteen olivat kielen funktionaaliset ominaisuudet kuten kirjastoidut kontrollirakenteet ja korkeamman asteen funktiot.

F#-ohjelmien rakennetta on helpompi muuttaa kuin C#-ohjelmien. Molemmissa kielissä on lähes samat työkalut ohjelmien arkkitehtuurin rakentamiseen. Molemmista kielistä löytyvät sekä nimiavaruudet että luokat. F#:ssa on näiden lisäksi moduulin käsite, jolla loogisesti yhteen kuuluvat ohjelman osat voidaan koota yhdeksi kokonaisuudeksi. Merkittävin F#-ohjelmien rakenteen muuttamista helpottava seikka on kuitenkin funktionaalisten periaatteiden mukaisesti toteutetun ohjelman tilattomuus ja funktioiden sivuvaikutuksettomuus.

C#-ohjelmista on helpompi jäljittää virheitä kuin F#-ohjelmista. Tässäkin tapauksessa C#:n paremmuus kuitenkin on kehitysympäristön hyvän tuen ansiota. Ohjelmistosuunnittelijat kokivat Visual Studion tuen F#-ohjelmien virheiden jäljittämiselle olevan heikko. Mielenkiintoista on kuitenkin se, että huonommasta virheiden jäljitettävyydestä huolimatta F#-ohjelmissa virheitä päätyy asiakkaalle asti vähemmän kuin C#-ohjelmissa.

Tyytyväisyyden osalta F# on selkeästi C#:ia parempi. Tulosten perusteella ohjelmistosuunnittelijat ovat huomattavasti tyytyväisempiä F#:iin. Kahdeksan yhdeksästä käyttäisi mieluummin F#:ia kuin C#:ia. F#:n käyttö koettiin hyödylliseksi siinäkin tapauksessa, että joutuisi tekemään lisätyötä C#-komponenttiin liittyäkseen. Tällaisessa tilanteessa kielten eri paradigmojen aiheuttamat erilaisuudet voidaan kapseloida käärimällä F#-komponentti C#-luokkaan niin, että paradigmojen erot peitetään komponentin käyttäjältä. Kääre on yksinkertainen toteuttaa ja aiheuttaa tämän takia vain vähän tai ei lainkaan virheitä.

LÄHTEET

[1] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 8 1978.

[2] Henri Bal and Dick Grune. *Programming Language Essentials*. Addison-Wesley, Chicago, IL., 1994. 270 p.

[3] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. An empirical study of programming language trends. *IEEE Software*, 22(3):72–78, 2005.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[5] Dave MacQueen. Standard ML SourceForge Project. [WWW], 2014. [viitattu 16.3.2014]. Saatavissa: `http://www.standardml.org`.

[6] Daniel Diaz. The gnu prolog web site. GNU. [WWW], 2013. [viitattu 6.2.2013]. Saatavissa: `http://www.gprolog.org`.

[7] ECMA International. *Standard ECMA-334 - C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), 4 edition, 6 2006.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley. 395 p., 1. edition, 1995.

[9] Google. Design elements. Google Developers. [WWW], 2013. [viitattu 24.1.2013]. Saatavissa: `https://developers.google.com/v8/design`.

[10] Ilkka Haikala and Jukka Märijärvi. *Ohjelmistotuotanto*. Korkeakoulu-sarja. Talentum Media Oy, 2006.

[11] Michael Hanus. Integration of declarative paradigms: Benefits and challenges. *SIGPLAN Notices*, 32(1):77–79, 1997.

[12] Maarit Harsu. *Ohjelmointikielet - Periaatteet, käsitteet, valintaperusteet*. Talentum, Helsinki, 2005. 322 s.

[13] Haskell.org. The haskell programming language. [WWW], 2014. [viitattu 15.3.2014]. Saatavissa: `http://www.haskell.org`.

[14] Ali Hodroj. Prolog.net. Prolog.NET. [WWW], 2013. [viitattu 6.2.2013]. Saatavissa: http://prolog.hodroj.net.

[15] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1984.

[16] IBM Corporation. Compiled versus interpreted languages. [WWW], 2013. [viitattu 24.1.2013]. Saatavissa: http://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp?topic=/com.ibm.zos.zappldev/zappldev_85.htm.

[17] IronPython Community. Ironpython. IronPython Community, 2013. [viitattu 10.12.2013]. Saatavissa: http://ironpython.net.

[18] IronRuby Community. Ironruby. IronRuby Community, 2013. [viitattu 10.12.2013]. Saatavissa: http://ironruby.net.

[19] John McCarthy. History of Lisp. [WWW], 2014. [viitattu 16.3.2014]. Saatavissa: http://www-formal.stanford.edu/jmc/history/lisp/lisp.html.

[20] Stephen Kaisler. *Software Paradigms*. Wiley, Hoboken, 2005.

[21] Thomas Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, IL., 3. edition, 1996.

[22] Kenneth Louden and Kenneth Lambert. *Programming languages: Principles and Practice*. Course Technology. 704 p., 3. edition, 2012.

[23] Microsoft. Common type system. Microsoft Developer Network. [WWW], 2013. [viitattu 10.12.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.110).aspx.

[24] Microsoft. Delegates (C# Programming Guide). Microsoft Developer Network. [WWW], 2013. [viitattu 16.12.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms173171.aspx.

[25] Microsoft. Language independence and language-independent components. Microsoft Developer Network. [WWW], 2013. [viitattu 10.12.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/12a7a7h3(v=vs.110).aspx.

[26] Microsoft. Overview of the .net framework. Microsoft Developer Network. [WWW], 2013. [viitattu 10.12.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx.

[27] Microsoft. Reflection (C# and Visual Basic). Microsoft Developer Network. [WWW], 2013. [viitattu 17.12.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms173183.aspx.

[28] Microsoft. Visual F#. Microsoft Developer Network. [WWW], 2013. [viitattu 17.12.2013]. Saatavissa: http://msdn.microsoft.com/en-us/library/dd233154.aspx.

[29] Microsoft. Automatic Generalization (F#). Microsoft Developer Network. [WWW], 2014. [viitattu 21.1.2014]. Saatavissa: http://msdn.microsoft.com/en-us/library/dd233183.aspx.

[30] Microsoft. Automatic Generalization (F#). Microsoft Developer Network. [WWW], 2014. [viitattu 22.1.2014]. Saatavissa: http://msdn.microsoft.com/en-us/library/hh225374.aspx.

[31] Microsoft. Delegates (F#). Microsoft Developer Network. [WWW], 2014. [viitattu 25.1.2014]. Saatavissa: http://msdn.microsoft.com/en-us/library/dd233206.aspx.

[32] Microsoft. Events (C# Programming Guide). Microsoft Developer Network. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: http://msdn.microsoft.com/en-us/library/awbftdfh.aspx.

[33] Microsoft. Using namespaces (C# programming guide). Microsoft Developer Network. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: http://msdn.microsoft.com/en-us/library/dfb3cx8s.aspx.

[34] Mozilla. Javascript. Mozilla Developer Network. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript?redirectlocale=en-US\&redirectslug=JavaScript.

[35] David Naiditch. Selecting a programming language for your project. *Aerospace and Electronic Systems Magazine, IEEE*, 14(9):11–14, 9 1999.

[36] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[37] Perl.org. The perl programming language. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: http://www.perl.org.

[38] Python Software Foundation. About python. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: http://www.python.org/about/.

[39] Python Software Foundation. Python programming language - official website. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: `http://www.python.org`.

[40] Research Software Limited. Miranda - A Non-strict Polymorphic Functional Language. [WWW], 2014. [viitattu 16.3.2014]. Saatavissa: `http://miranda.org.uk`.

[41] Peter van Roy. Programming paradigms for dummies: What every programmer should know. In G. Assayag, A. Gerzso, and IRCAM (Research institute : France), editors, *New Computational Paradigms for Computer Music*, Collection Musique/sciences. Editions Delatour France, 2009.

[42] Peter van Roy and Seif Haridi. *Concepts, Techniques and Models of Computer Programming.* Cambridge, Massachusetts London England. The MIT Press, 2004.

[43] Ruby community. Ruby. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: `https://www.ruby-lang.org/en/`.

[44] A.J. Sanchez-Ruiz and E.P. Glinert. Multilanguage programming: an automatic-type-mapping approach. In *Computer Software and Applications Conference, 1992. COMPSAC '92. Proceedings., Sixteenth Annual International*, pages 57 –62, 9 1992.

[45] Michael L. Scott. *Programming Language Pragmatics.* Kaufmann, Morgan. 915 p., 2. edition, 2012.

[46] SWI Prolog. Swi-prolog's home. SWI Prolog. [WWW], 2013. [viitattu 6.2.2013]. Saatavissa: `http://www.swi-prolog.org`.

[47] The PHP Group. php. [WWW], 2014. [viitattu 2.1.2014]. Saatavissa: `http://php.net`.

[48] United States Government. Package Specifications and Declarations. Ada '83 Language Reference Manual. [WWW], 2014. [viitattu 23.2.2014]. Saatavissa: `http://archive.adaic.com/standards/83lrm/html/lrm-07-02.html`.

[49] Steve Vinoski. Multilanguage programming. *Internet Computing, IEEE*, 12(3):83 –85, 2008.

[50] Peter Wegner. Guest editor's introduction to special issue of computing surveys on programming language paradigms. *ACM Comput. Surv.*, 21(3):253–258, 1989.

[51] Yin Zhu. A functional red black tree with dynamic order statistics in F#. [WWW], 2014. [viitattu 17.2.2014]. Saatavissa: `http://fdatamining.blogspot.fi/2011/09/functional-red-black-tree-with-dynamic.html`.

[52] Marvin Zelkowitz. Programming languages, 2012.

A. ESIMERKKIOHJELMAN LÄHDEKOODI

A.1 Testiohjelma

Ohjelma A.1: Testiohjelma, joka suorittaa määritellyt testitapaukset.

```
1  open TestProgram.TestCases
2
3  [<EntryPoint>]
4  let main argv =
5      try
6          test1()
7          test2()
8          test3()
9          test4()
10         printf "Test␣program␣passed!␣\nPress␣any␣key␣to␣continue..."
11     with
12     | e as Exception ->
13         printf "Exception!␣Messgae:␣%s" e.Message
14
15     System.Console.ReadKey() |> ignore
16     0 // return an integer exit code
```

A.2 Testitapaukset

Ohjelma A.2: Moduuli, jossa testitapaukset on määritelty.

```
1  namespace TestProgram
2
3  module TestCases =
4      open Cs_RBTree
5      open Fs_RBTree.RBTree
6
7          // Lisätäänn punamustan puun \Fs toteutukseen tuhat satunnaislukua
8          // punamustaan puuhun ja poistetaan ne.
9      let test1() =
10         let testSize = 1000
11         let rnd = new System.Random(42)
12         let testValues = List.init testSize (fun _ -> rnd.Next(0,1000)) |> Set.
               ↪ ofList
13         let _,tree =
14             Set.fold (fun (count,tree : FsRBTree<int>) value ->
15                             if tree.contains value |> not then
16                                 let tree = tree.insert value
17                                 if tree.size <> count + 1 then
18                                     failwith "Added␣item␣count␣and␣tree␣size␣
                                           ↪ mismatch!"
19                                 tree.Validate() |> ignore
20                                 (count + 1, tree)
```

```
21                          else (count,tree)) (0,Fs_RBTree.RBTree.empty)
                                ↪ testValues
22          let size = tree.size
23          let _ = Set.fold (fun (count,tree : FsRBTree<int>) value ->
24                          let newTree = tree.delete value
25                          if newTree.size <> count then
26                              failwith "Item␣count␣and␣tree␣size␣mismatch␣after␣
                                    ↪ deletion!"
27                          newTree.Validate()
28                          count - 1,newTree) (tree.size - 1,tree) testValues
29          ()
30
31          // Lisätäänn punamustan puun \Fs toteutukseen tuhat lukua järjestyksessä
32          // ja poistetaan ne.
33      let test2() =
34          let testSize = 1000
35          let testValues = [for i in 1..testSize do yield i] |> Set.ofList
36          let _,tree =
37              Set.fold (fun (count,tree : FsRBTree<int>) value ->
38                          if tree.contains value |> not then
39                              let tree = tree.insert value
40                              if tree.size <> count + 1 then
41                                  failwith "Added␣item␣count␣and␣tree␣size␣
                                        ↪ mismatch!"
42                              tree.Validate() |> ignore
43                              (count + 1, tree)
44                          else (count,tree)) (0,Fs_RBTree.RBTree.empty)
                                ↪ testValues
45
46          let _ = Set.fold (fun (count,tree : FsRBTree<int>) value ->
47                          let newTree = tree.delete value
48                          if newTree.size <> count then
49                              failwith "Item␣count␣and␣tree␣size␣mismatch␣after␣
                                    ↪ deletion!"
50                          newTree.Validate()
51                          count - 1,newTree) (tree.size - 1,tree) testValues
52          ()
53
54          // Lisätään punamustan puun \Cs toteutukseen tuhat satunnaislukua
55          // punamustaan puuhun ja poistetaan ne.
56      let test3() =
57          let testSize = 1000
58          let rnd = new System.Random(42)
59          let testValues = List.init testSize (fun _ -> rnd.Next(0,1000)) |> Set.
                ↪ ofList
60          let tree = new CsRBTree<TestObject<int>>()
61
62          let _ = Set.fold (fun count value ->
63                          let testValue = new TestObject<int>(value)
64                          if not <| tree.Contains(testValue) then
65                              tree.Insert(testValue)
66                              if tree.Size <> count + 1 then
67                                  failwith "Item␣count␣and␣tree␣size␣mismatch␣
                                        ↪ after␣insertion!"
68                              count + 1
69                          else count
70                          ) 0 testValues
71
72          let _ = Set.fold (fun count value ->
```

```
73                              let testValue = new TestObject<int>(value)
74                              if tree.Contains(testValue) then
75                                  tree.Delete(testValue) |> ignore
76                                  if tree.Size <> count - 1 then
77                                      failwith "Item␣count␣and␣tree␣size␣mismatch␣
                                            ↪ after␣deletion!"
78                                  count - 1
79                              else count
80                              ) tree.Size testValues
81          ()
82
83          // Lisätään punamustan puun \Cs toteutukseen tuhat lukua järjestyksessä
84          // ja poistetaan ne.
85      let test4() =
86          let testSize = 1000
87          let testValues = [for i in 1..testSize do yield i] |> Set.ofList
88          let tree = new CsRBTree<TestObject<int>>()
89
90          let _ = Set.fold (fun count value ->
91                              let testValue = new TestObject<int>(value)
92                              if not <| tree.Contains(testValue) then
93                                  tree.Insert(testValue)
94                                  if tree.Size <> count + 1 then
95                                      failwith "Item␣count␣and␣tree␣size␣mismatch␣
                                            ↪ after␣insertion!"
96                                  count + 1
97                              else count
98                              ) 0 testValues
99
100         let _ = Set.fold (fun count value ->
101                             let testValue = new TestObject<int>(value)
102                             if tree.Contains(testValue) then
103                                 tree.Delete(testValue) |> ignore
104                                 if tree.Size <> count - 1 then
105                                     failwith "Item␣count␣and␣tree␣size␣mismatch␣
                                            ↪ after␣deletion!"
106                                 count - 1
107                             else count
108                             ) tree.Size testValues
109         ()
```

A.3 Punamusta puu: C#-toteutus

Ohjelma A.3: Punamustan puun toteuttava luokka.

```
1  using System;
2
3  namespace Cs_RBTree
4  {
5     public class CsRBTree<T> where T : class, IComparable
6     {
7        private enum Color { Red, Black };
8
9        public CsRBTree ()
10       {
11          Root = Node<T>.Sentinel;
12       }
13
14       private Node<T> Root { get; set; }
```

```
15
16          public int Depth
17          {
18              get
19              {
20                  return Root.GetDepth ();
21              }
22          }
23
24          public int Size
25          {
26              get
27              {
28                  return Root.GetSize ();
29              }
30          }
31
32          public bool Contains (T x)
33          {
34              return Root.Contains (x);
35          }
36
37          #region Insert
38
39          public void Insert (T e)
40          {
41              Node<T> z = new Node<T> (e);
42              Node<T> y = Node<T>.Sentinel;
43              Node<T> x = Root;
44
45              while (x != Node<T>.Sentinel)
46              {
47                  y = x;
48                  if (z.Element.CompareTo (x.Element) < 0)
49                  {
50                      x = x.LeftChild;
51                  }
52                  else
53                  {
54                      x = x.RightChild;
55                  }
56              }
57              z.Parent = y;
58              if (y == Node<T>.Sentinel)
59              {
60                  Root = z;
61              }
62              else if (z.Element.CompareTo (y.Element) < 0)
63              {
64                  y.LeftChild = z;
65              }
66              else
67              {
68                  y.RightChild = z;
69              }
70              z.LeftChild = Node<T>.Sentinel;
71              z.RightChild = Node<T>.Sentinel;
72              z.Color = Node<T>.NodeColor.Red;
73              InsertFixUp (z);
```

```
74        }
75
76        private void InsertFixUp (Node<T> z)
77        {
78           while (z.Parent.Color == Node<T>.NodeColor.Red)
79           {
80              if (z.Parent == z.Parent.Parent.LeftChild)
81              {
82                 Node<T> y = z.Parent.Parent.RightChild;
83                 if (y.Color == Node<T>.NodeColor.Red)
84                 {
85                    z.Parent.Color = Node<T>.NodeColor.Black;
86                    y.Color = Node<T>.NodeColor.Black;
87                    z.Parent.Parent.Color = Node<T>.NodeColor.Red;
88                    z = z.Parent.Parent;
89                 }
90                 else
91                 {
92                    if (z == z.Parent.RightChild)
93                    {
94                       z = z.Parent;
95                       LeftRotate (z);
96                    }
97                    z.Parent.Color = Node<T>.NodeColor.Black;
98                    z.Parent.Parent.Color = Node<T>.NodeColor.Red;
99                    RightRotate (z.Parent.Parent);
100                }
101             }
102             else
103             {
104                Node<T> y = z.Parent.Parent.LeftChild;
105                if (y.Color == Node<T>.NodeColor.Red)
106                {
107                   z.Parent.Color = Node<T>.NodeColor.Black;
108                   y.Color = Node<T>.NodeColor.Black;
109                   z.Parent.Parent.Color = Node<T>.NodeColor.Red;
110                   z = z.Parent.Parent;
111                }
112                else
113                {
114                   if (z == z.Parent.LeftChild)
115                   {
116                      z = z.Parent;
117                      RightRotate (z);
118                   }
119                   z.Parent.Color = Node<T>.NodeColor.Black;
120                   z.Parent.Parent.Color = Node<T>.NodeColor.Red;
121                   LeftRotate (z.Parent.Parent);
122                }
123             }
124          }
125          Root.Color = Node<T>.NodeColor.Black;
126       }
127
128       #endregion
129
130       #region Delete
131
132       public T Delete (T e)
```

```
133         {
134             Node<T> z = Root.Find (e);
135 
136             if (z == null)
137             {
138                 throw new ArgumentException ("Value␣not␣found.");
139             }
140 
141             Node<T> y = Node<T>.Sentinel;
142             if (z.LeftChild == Node<T>.Sentinel || z.RightChild == Node<T>.Sentinel)
143             {
144                 y = z;
145             }
146             else
147             {
148                 y = GetSuccessor (z);
149             }
150 
151             Node<T> x = Node<T>.Sentinel;
152             if (y.LeftChild != Node<T>.Sentinel)
153             {
154                 x = y.LeftChild;
155             }
156             else
157             {
158                 x = y.RightChild;
159             }
160 
161             x.Parent = y.Parent;
162 
163             if (y.Parent == Node<T>.Sentinel)
164             {
165                 Root = x;
166             }
167             else if (y == y.Parent.LeftChild)
168             {
169                 y.Parent.LeftChild = x;
170             }
171             else
172             {
173                 y.Parent.RightChild = x;
174             }
175 
176             if (y != x)
177             {
178                 z.Element = y.Element;
179             }
180 
181             if (y.Color == Node<T>.NodeColor.Black)
182             {
183                 DeleteFixup (x);
184             }
185             return y.Element;
186         }
187 
188         private Node<T> GetSuccessor (Node<T> x)
189         {
190             if (x.RightChild != Node<T>.Sentinel)
191             {
```

```
192             return GetMinimum (x.RightChild);
193          }
194
195          Node<T> y = x.Parent;
196          while (y != Node<T>.Sentinel && x == y.RightChild)
197          {
198             x = y;
199             y = y.Parent;
200          }
201          return y;
202       }
203
204       private Node<T> GetMinimum (Node<T> x)
205       {
206          while (x.LeftChild != Node<T>.Sentinel)
207          {
208             x = x.LeftChild;
209          }
210          return x;
211       }
212
213       private void DeleteFixup (Node<T> x)
214       {
215          while (x != Root && x.Color == Node<T>.NodeColor.Black)
216          {
217             if (x == x.Parent.LeftChild)
218             {
219                Node<T> w = x.Parent.RightChild;
220                if (w.Color == Node<T>.NodeColor.Red)
221                {
222                   w.Color = Node<T>.NodeColor.Black;
223                   x.Parent.Color = Node<T>.NodeColor.Red;
224                   LeftRotate (x.Parent);
225                   w = x.Parent.RightChild;
226                }
227                if (w.LeftChild.Color == Node<T>.NodeColor.Black && w.RightChild.
                       ↪ Color == Node<T>.NodeColor.Black) {
228                   w.Color = Node<T>.NodeColor.Red;
229                   x = x.Parent;
230                }
231                else
232                {
233                   if (w.RightChild.Color == Node<T>.NodeColor.Black)
234                   {
235                      w.LeftChild.Color = Node<T>.NodeColor.Black;
236                      w.Color = Node<T>.NodeColor.Black;
237                      RightRotate (w);
238                      w = x.Parent.RightChild;
239                   }
240                   w.Color = x.Parent.Color;
241                   x.Parent.Color = Node<T>.NodeColor.Black;
242                   w.RightChild.Color = Node<T>.NodeColor.Black;
243                   LeftRotate (x.Parent);
244                   x = Root;
245                }
246             }
247             else
248             {
249                Node<T> w = x.Parent.LeftChild;
```

```
250                 if (w.Color == Node<T>.NodeColor.Red)
251                 {
252                     w.Color = Node<T>.NodeColor.Black;
253                     x.Parent.Color = Node<T>.NodeColor.Red;
254                     RightRotate (x.Parent);
255                     w = x.Parent.LeftChild;
256                 }
257                 if (w.RightChild.Color == Node<T>.NodeColor.Black && w.LeftChild.
                        ↪ Color == Node<T>.NodeColor.Black)
258                 {
259                     w.Color = Node<T>.NodeColor.Red;
260                     x = x.Parent;
261                 }
262                 else
263                 {
264                     if (w.LeftChild.Color == Node<T>.NodeColor.Black)
265                     {
266                         w.RightChild.Color = Node<T>.NodeColor.Black;
267                         w.Color = Node<T>.NodeColor.Black;
268                         LeftRotate (w);
269                         w = x.Parent.LeftChild;
270                     }
271                     w.Color = x.Parent.Color;
272                     x.Parent.Color = Node<T>.NodeColor.Black;
273                     w.LeftChild.Color = Node<T>.NodeColor.Black;
274                     RightRotate (x.Parent);
275                     x = Root;
276                 }
277             }
278         }
279         x.Color = Node<T>.NodeColor.Black;
280     }
281
282     #endregion
283
284     private void LeftRotate (Node<T> x)
285     {
286         Node<T> y = x.RightChild;
287         x.RightChild = y.LeftChild;
288         if (y.LeftChild != Node<T>.Sentinel)
289         {
290             y.LeftChild.Parent = x;
291         }
292         y.Parent = x.Parent;
293         if (x.Parent == Node<T>.Sentinel)
294         {
295             Root = y;
296         }
297         else if (x == x.Parent.LeftChild)
298         {
299             x.Parent.LeftChild = y;
300         }
301         else
302         {
303             x.Parent.RightChild = y;
304         }
305         y.LeftChild = x;
306         x.Parent = y;
307     }
```

```
308
309         private void RightRotate (Node<T> x)
310         {
311            Node<T> y = x.LeftChild;
312            x.LeftChild = y.RightChild;
313            if (y.RightChild != Node<T>.Sentinel)
314            {
315               y.RightChild.Parent = x;
316            }
317            y.Parent = x.Parent;
318            if (x.Parent == Node<T>.Sentinel)
319            {
320               Root = y;
321            }
322            else if (x == x.Parent.RightChild)
323            {
324               x.Parent.RightChild = y;
325            }
326            else
327            {
328               x.Parent.LeftChild = y;
329            }
330            y.RightChild = x;
331            x.Parent = y;
332         }
333
334         public void ValidateTree ()
335         {
336            if (Root.Color == Node<T>.NodeColor.Red)
337            {
338               throw new Exception ("Red␣root!");
339            }
340
341            Root.ValidateNode ();
342         }
343      }
344   }
```

Ohjelma A.4: Punamustan puun solmua esittävä luokka.

```
1  using System;
2
3  namespace Cs_RBTree
4  {
5     public class Node<T> where T : class, IComparable
6     {
7        public enum NodeColor { Red, Black };
8
9        public static readonly Node<T> Sentinel = new Node<T> (null) { Color = Node<T
             ↪ >.NodeColor.Black };
10
11       public Node (T e)
12       {
13          Parent = Sentinel;
14          LeftChild = Sentinel;
15          RightChild = Sentinel;
16          Element = e;
17       }
18
```

```
19        public NodeColor Color { get; set; }
20
21        public Node<T> Parent { get; set; }
22
23        public Node<T> LeftChild { get; set; }
24
25        public Node<T> RightChild { get; set; }
26
27        public T Element { get; set; }
28
29        public int GetDepth ()
30        {
31           if (this == Sentinel)
32           {
33              return 0;
34           }
35
36           return 1 + Math.Max (LeftChild.GetDepth (), RightChild.GetDepth ());
37        }
38
39        public int GetSize ()
40        {
41           if (this == Sentinel)
42           {
43              return 0;
44           }
45
46           return 1 + LeftChild.GetSize () + RightChild.GetSize ();
47        }
48
49        internal int ValidateNode ()
50        {
51           int leftBlackCount = 0;
52           int rightBlackCount = 0;
53
54           if (Color == NodeColor.Red)
55           {
56              if (LeftChild != null && LeftChild.Color == NodeColor.Red)
57              {
58                 throw new Exception ("Double␣red!");
59              }
60              else
61              {
62                 leftBlackCount = LeftChild.ValidateNode ();
63              }
64
65              if (RightChild != null && RightChild.Color == NodeColor.Red)
66              {
67                 throw new Exception ("Double␣red!");
68              }
69              else
70              {
71                 rightBlackCount = RightChild.ValidateNode ();
72              }
73           }
74
75           if (leftBlackCount != rightBlackCount)
76           {
77              throw new Exception ("Black␣node␣count␣differs␣in␣left␣and␣right␣
```

```
                ↪ subtrees!");
78          }
79
80          return (leftBlackCount + rightBlackCount) + (Color == NodeColor.Black ? 1
                ↪ : 0);
81       }
82
83       internal bool Contains (T x)
84       {
85          if (this == Sentinel)
86          {
87             return false;
88          }
89          else if (Element == x)
90          {
91             return true;
92          }
93          else
94          {
95             return LeftChild.Contains (x) || RightChild.Contains (x);
96          }
97       }
98
99       internal Node<T> Find (object e)
100      {
101         if (this == Sentinel)
102         {
103            return null;
104         }
105
106         if (Element.CompareTo (e) < 0)
107         {
108            return LeftChild.Find (e);
109         }
110         else if (Element.CompareTo (e) > 0)
111         {
112            return RightChild.Find (e);
113         }
114
115         return this;
116      }
117   }
118}
```

Ohjelma A.5: Testeissä käytetty kääre, jonka avulla myös perustietotyyppejä voidaan tallentaa punamustan puun C#-toteutukseen.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Cs_RBTree
8  {
9     public class TestObject<T> : IComparable where T : IComparable
10    {
11       public TestObject (T init)
12       {
```

```
13          Element = init;
14       }
15
16       public T Element { get; set; }
17
18       #region IComparable Members
19
20       public int CompareTo (object obj)
21       {
22          if (obj == null)
23          {
24             throw new ArgumentNullException ();
25          }
26
27          var objB = obj as TestObject<T>;
28          if (objB == null)
29          {
30             throw new ArgumentException ("Invalid␣argument␣type.");
31          }
32
33          return Element.CompareTo (objB.Element);
34       }
35
36       #endregion
37    }
38 }
```

A.4 Punamusta puu: F#-toteutus

Ohjelma A.6: Punamustan puun F#-toteutus kokonaisuudessaan.

```
1  namespace Fs_RBTree
2
3  module RBTree =
4
5      type Color =
6          | Red
7          | Black
8          | DoubleBlack
9
10     type FsRBTree<'t when 't : comparison> =
11         | Leaf of Color
12         | Node of Color * FsRBTree<'t> * 't * FsRBTree<'t>
13         with
14
15             member this.isDoubleBlack =
16                 match this with
17                 | Leaf(DoubleBlack)
18                 | Node(DoubleBlack,_,_,_) -> true
19                 | _ -> false
20
21             member this.isBlack =
22                 match this with
23                 | Leaf(Black)
24                 | Node(Black,_,_,_) -> true
25                 | _ -> false
26
27             member this.blackify =
28                 match this with
```

```
29              | Leaf _ -> Leaf Black
30              | Node(_,l,e,r) -> Node(Black,l,e,r)
31
32          member this.contains x =
33              match this with
34              | Leaf _ -> false
35              | Node(_,l,y,r) when x < y -> l.contains x
36              | Node(_,l,y,r) when x > y -> r.contains x
37              | _ -> true
38
39          member this.insert x =
40              let balance tree =
41                  match tree with
42                  | Node(Black,Node(Red,Node(Red,a,x,b),y,c),z,d)
43                  | Node(Black,Node(Red,a,x,Node(Red,b,y,c)),z,d)
44                  | Node(Black,a,x,Node(Red,Node(Red,b,y,c),z,d))
45                  | Node(Black,a,x,Node(Red,b,y,Node(Red,c,z,d))) ->
46                      Node(Red,Node(Black,a,x,b),y,Node(Black,c,z,d))
47                  | _ -> tree
48
49              let rec ins tree =
50                  match tree with
51                  | Leaf _ -> Node(Red,Leaf Black,x,Leaf Black)
52                  | Node(color,l,y,r) when x < y -> balance (Node(color,(ins l),y
                        ↪ ,r))
53                  | Node(color,l,y,r) when x > y -> balance (Node(color,l,y,(ins
                        ↪ r)))
54                  | _ -> tree
55
56              match ins this with
57              | Node(_,l,e,r) -> Node(Black,l,e,r)
58              | this -> this
59
60          member this.depth =
61              match this with
62              | Leaf _ -> 0
63              | Node(_,l,_,r) -> 1 + max l.depth r.depth
64
65          member this.size =
66              match this with
67              | Leaf _ -> 0
68              | Node(_,l,_,r) -> 1 + l.size + r.size
69
70          member this.at n =
71              let rec at' n tree =
72                  match tree with
73                  | Leaf _ -> None
74                  | Node(_,l,e,_) when l.size + 1 = n -> Some e
75                  | Node(_,l,_,_) when l.size + 1 > n -> at' (n - l.size - 1) l
76                  | Node(_,l,_,r) -> at' (n - l.size - 1) r
77              if n >= this.size || n < 0 then
78                  None
79              else
80                  at' n this
81
82          member this.min =
83              match this with
84              | Leaf _ -> failwith "Tree␣is␣empty."
85              | Node(_,Leaf _,e,_) -> e
```

```
86                  | Node(_,l,_,_) -> l.min
87
88              member this.delete x =
89                  let rec balance tree =
90                      match tree with
91                      | Node(Black,pl,pe,Node(Red,rl,re,rr))
92                          when pl.isDoubleBlack ->
93                          balance (Node(Black,balance (Node(Red,pl,pe,rl)),re,rr))
94                      | Node(Red,pl,pe,Node(Black,rl,re,rr))
95                          when pl.isDoubleBlack && rl.isBlack && rr.isBlack ->
96                          Node(Black,pl.blackify,pe,Node(Red,rl,re,rr))
97                      | Node(Black,pl,pe,Node(Black,rl,re,rr))
98                          when pl.isDoubleBlack && rl.isBlack && rr.isBlack ->
99                          Node(DoubleBlack,pl.blackify,pe,Node(Red,rl,re,rr))
100                     | Node(DoubleBlack,pl,pe,Node(Black,rl,re,rr))
101                         when pl.isDoubleBlack && rl.isBlack && rr.isBlack ->
102                         Node(Red,pl.blackify,pe,Node(Red,rl,re,rr))
103                     | Node(pc,pl,pe,Node(Black,Node(_,rll,rle,rlr),re,rr))
104                         when pl.isDoubleBlack && rr.isBlack ->
105                         balance (Node(pc,pl,pe,
106                                     Node(Black,rll,rle,Node(Red,rll,re,rr))))
107                     | Node(pc,pl,pe,Node(Black,rl,re,rr))
108                         when pl.isDoubleBlack ->
109                         Node(pc,Node(Black,pl.blackify,pe,rl),re,rr.blackify)
110                     | Node(Black,Node(Red,ll,le,lr),pe,pr)
111                         when pr.isDoubleBlack ->
112                         balance (Node(Black,ll,le,balance (Node(Red,lr,pe,pr))))
113                     | Node(Red,Node(Black,ll,le,lr),pe,pr)
114                         when pr.isDoubleBlack && ll.isBlack && lr.isBlack ->
115                         Node(Black,Node(Red,ll,le,lr),pe,pr.blackify)
116                     | Node(Black,Node(Black,ll,le,lr),pe,pr)
117                         when pr.isDoubleBlack && ll.isBlack && lr.isBlack ->
118                         Node(DoubleBlack,Node(Red,ll,le,lr),pe,pr.blackify)
119                     | Node(DoubleBlack,Node(Black,ll,le,lr),pe,pr)
120                         when pr.isDoubleBlack && ll.isBlack && lr.isBlack ->
121                         Node(Red,Node(Red,ll,le,lr),pe,pr.blackify)
122                     | Node(pc,Node(Black,ll,le,Node(_,lrl,lre,lrr)),pe,pr)
123                         when pr.isDoubleBlack && ll.isBlack ->
124                         balance (Node(pc,
125                                     Node(Black,Node(Red,ll,le,lrl),lre,lrr),pe,pr))
126                     | Node(pc,Node(Black,ll,le,lr),pe,pr) when pr.isDoubleBlack ->
127                         Node(pc,ll.blackify,le,Node(Black,lr,pe,pr.blackify))
128                     | t -> t
129
130                 let rec del tree =
131                     match tree with
132                     | Node(c,l,e,r) when x < e -> balance (Node(c,(del l),e,r))
133                     | Node(c,l,e,r) when x > e -> balance (Node(c,l,e,(del r)))
134                     | Node(Black,Leaf Black,e,Leaf Black) when x = e ->
135                         Leaf DoubleBlack
136                     | Node(Red,Leaf Black,e,Leaf Black) when x = e ->
137                         Leaf Black
138                     | Node(DoubleBlack,Leaf Black,e,Leaf Black) when x = e ->
139                         Leaf Red
140                     | Node(c,Node(cc,cl,ce,cr),e,Leaf _)
141                     | Node(c,Leaf _,e,Node(cc,cl,ce,cr)) when x = e ->
142                         match c,cc with
143                         | Red, Black
144                         | Black, Red -> Node(Black,cl,ce,cr)
```

```
145                         | Black, Black -> Node(DoubleBlack,cl,ce,cr)
146                         | _,_ -> Node(Red,cl,ce,cr)
147                     | Node(c,l,e,r) when x = e ->
148                         let newValue = r.min
149                         balance (Node(c,l,newValue,(r.delete newValue)))
150                     | t -> t
151                 match del this with
152                 | Node(_,l,e,r) -> Node(Black,l,e,r)
153                 | t -> t
154 
155             member this.blackHeight =
156                 let rec height count tree =
157                     match tree with
158                     | Leaf _ -> count + 1
159                     | Node(Red,l,_,r) -> List.max [(height count l); (height count
                        ↪ r)]
160                     | Node(_,l,_,r) -> List.max [(height (count + 1) l); (height (
                        ↪ count + 1) r)]
161                 height 0 this
162 
163             // For debug purposes only!
164             member this.Validate() =
165                 let rec validateReds tree =
166                     match tree with
167                     | Leaf _ -> ()
168                     | Node(c,l,e,r) ->
169                         match l,r with
170                         | Node(Red,_,_,_),_
171                         | _,Node(Red,_,_,_) when c = Red -> failwith "Double␣red!"
172                         | _,_ ->
173                             validateReds l
174                             validateReds r
175 
176                 match this with
177                 | Node(Red,_,_,_) -> failwith "Root␣must␣be␣black"
178                 | Node(_,l,_,r) ->
179                     validateReds this
180                     let leftBlackHeight = l.blackHeight
181                     let rightBlackHeight = r.blackHeight
182                     if leftBlackHeight <> rightBlackHeight then
183                         failwith "Blach␣height␣mismatch"
184                 | _ -> ()
185 
186     let empty<'t when 't : comparison> : FsRBTree<'t> = Leaf Black
```